Table of Contents

Table of Contents

Table of Contents

Table of Contents

Preface

When I first mentioned to a colleague of mine that I was writing a book on JavaTM security, he immediately starting asking me questions about firewalls and Internet DMZs. Another colleague overheard us and started asking about electronic commerce, which piqued the interest of a third colleague who wanted to hear all about virtual private networks. All this was interesting, but what I really wanted to talk about was how a Java applet could be allowed to read a file.

Such is the danger of anything with the word "security" in its title: security is a broad topic, and everyone has his or her own notion of what security means. Complicating this issue is the fact that Java security and network security (including Internet security) are complementary and sometimes overlapping topics: you can send encrypted data over the network with Java, or you can set up a virtual private network that encrypts all your network traffic and remove the need for encryption within your Java programs.

This is a book about security from the perspective of a Java program. In this book, we discuss the basic platform features of Java that provide security −− the class loader, the bytecode verifier, the security manager −− and we discuss recent additions to Java that enhance this security model −− digital signatures, security providers, and the access controller. The ideas in this book are meant to provide an understanding of the architecture of Java's security model and how that model can be used (both programmatically and administratively).

Who Should Read This Book?

This book is intended primarily for programmers who want to write secure Java applications. Much of the book is focused on various APIs within Java that provide security; we discuss how those APIs are used by a standard Java 2 Standard Edition implementation. This includes both how these APIs may be used from within your own application and how they may be used from within an applet that runs in the Java Plug−in. The Java Plug−in comes with all versions of the Java 2 platform and allows you to run applets in a virtual machine that is decoupled from the browser, which allows you to have full Java 2 support in most current browsers, from Internet Explorer (version 3 and higher) to Netscape Navigator (version 4 and higher) to Opera. In later versions of Navigator and Opera, the Plug−in is the only supported virtual machine. This is particularly important with respect to security since no browser supports the Java 2 security model in its embedded virtual machine.

For the end user or system administrator who is interested in Java security, this book will provide knowledge of the facilities provided by the basic Java platform and how those facilities are used by Java applications and applets within the Java Plug−in. We do not delve into the specific security features of Java−enabled browsers, although we do point out along the way which security features of Java are subject to change by the companies that provide Java−enabled browsers. Hence, end users and system administrators can read this book (and skip over many of the programming examples) to gain an understanding of the fundamental security features of the Java platform, and they can understand from each of its parts how the security features might be administered. This is particularly true for end users and administrators who are interested in assessing the risk of using Java: we give full details of the implementation of Java's security model so that you can program within that model (and adjust it if necessary); we give you a deep understanding of how the model works so you can assess for yourself whether or not Java meets your definition of security.

From a programming perspective, we assume that developers who read this book have a good knowledge of how to program in Java, and in particular how to write Java applications. When we discuss advanced security features and cryptographic algorithms, we do so assuming that the programmer is primarily interested in using the API to perform certain tasks. Hence, we explain at a rudimentary level what a digital signature is and how it is created and used, but we do not explain the cryptographic theory behind a digital signature or prove that a digital signature is secure. For developers who are sufficiently versed in these matters, we also show how the APIs may be extended to support new types of cryptographic algorithms, but again we leave the mathematics and rigorous definitions of cryptography for another book.

Versions Used in This Book

This book is based on Java 2 Standard Edition, version 1.3 (alternately called simply 1.3). The security model of the Java 2 platform is radically different from the model of Java 1. Many basic security interfaces such as the access controller were introduced in Java 2, and other APIs went through significant changes between Java 1.1 and Java 2. On the other hand, there were few changes between Java 2 version 1.2 and 1.3; most of the information we discuss is applicable to 1.2 as well.

We also discuss three Java extensions in this book: version 1.2.1 of the Java Cryptography Extension, version 1.0.2 of the Java Secure Sockets Extension, and version 1.0 of the Java Authentication and Authorization Service. These extensions all rely on version 1.3 of the Java 2 platform.

Code examples used in this book are available from the O'Reilly web site located at http://www.oreilly.com/catalog/javasec2/.

Conventions Used in This Book

`Constant width font` is used for:

- Code examples
- Class, variable, and method names within the text

Italicized font is used for:

- Filenames
- Host and domain names
- URLs

When a new method or class is introduced, its definition will appear beginning with italicized text like this:

public void checkAccess(Thread t)

Check whether the current thread is allowed to modify the state of the parameter thread.

Command Conventions

There are some examples of commands scattered through the book, especially in sections and appendices that deal with administration. By convention, most examples are shown as they would be executed on a Unix system, e.g.:

```
piccolo% keytool −export −alias sdo −file /tmp/sdo.cer
Enter keystore password: ******
Certificate stored in file </tmp/sdo.cer>
```

In these examples, the text typed by the user or administrator is always shown in **bold font**; the remaining text is output from the command (the string piccolo% indicates a command prompt). On other systems, the names of the files would have to be changed to conform to that system (e.g., C:\sdo.cer for a Microsoft Windows system). Keep in mind, however, that the command−line arguments often specify a URL rather than a filename, in which case you must use forward slashes. In that case, the argument is often the same, although on Microsoft Windows systems you must specify a drive: the Unix directory *file:///files/sdo/* is rendered on Microsoft Windows as *file:/C:/files/sdo/*. When an argument requires a URL, we always specify the protocol to distinguish it from a filename, even though tools will often accept the string without a protocol.

However, note that while Microsoft Windows systems often use a forward−slash (/) for command−line options, Java tools (even on those systems) universally use a hyphen (−) to indicate command−line options. In these examples, then, only the file and URL names are different between platforms.

Code Conventions

The code examples in this book (and in the online samples) are organized by chapter. Each class belongs to a package based on the chapter when it is introduced; e.g., the class Test from Chapter 3, is in the package javasec.samples.ch03. When you unpack the online code, you'll end up with a single directory (javasec), from which the remaining directories and source files descend.

There are two simple ways of proceeding. The first is to remain in the directory where you unpacked the sources, not set your classpath, and reference everything by full package name. Hence, to compile and run the Test class from Chapter 3, you'd execute these commands:

```
piccolo% javac javasec/samples/ch03/Test.java
piccolo% java javasec.samples.ch03.Test
Your account number is 0001 0002 0003 0004
```

Alternately, you can work in the directory containing the source and set your classpath as follows:

```
piccolo% javac −classpath ../../.. Test.java
piccolo% java −classpath ../../.. javasec.samples.ch03.Test
Your account number is 0001 0002 0003 0004
```

When required for space, commands may be continued on multiple lines, in which case a backslash character is used:

```
piccolo% java −classpath ../../.. javasec.samples.ch09.PrintCert \
              /files/sdo/foo/bar/very/long/command
```

Commands that appear like this should be typed on one line or typed on multiple lines using whatever escape character is appropriate for your system (e.g., the backslash for a Unix system).

Organization of This Book

This book is organized in a bottom−up fashion: we begin with the very low−level aspects of Java security and then proceed to the more advanced features.

Chapter 1
: This chapter gives an overview of the security model (the Java sandbox) used in Java applications and sets the stage for the rest of the book.

Chapter 2
: This chapter discusses the parameters of the default sandbox and how the sandbox may be changed administratively. It provides instructions for end users and administrators on how to set up Java security policies (including the use of `policytool`) and introduces the concepts by which these policies are implemented.

Chapter 3

This chapter discusses the memory protections built into the Java language, how those protections provide a measure of security, and how they are enforced by the bytecode verifier.

Chapter 4

This chapter discusses the security manager, which is the primary interface to application−level security in Java. The security manager is responsible for arbitrating access to all local resources: files, the network, printers, etc.

Chapter 5

The access controller is the basis for security manager implementations in Java 2. This chapter discusses how to use the access controller to achieve fine−grained levels of security in your application.

Chapter 6

This chapter discusses the class loader, which is the class that reads in Java class files and turns them into classes. From a security perspective, the class loader is important in determining where classes originated and whether or not they were digitally signed (and if so, by whom), so the topic of class loaders appears throughout this book.

Chapter 7

This chapter provides an overview to the cryptographic algorithms of the Java security package. It provides a background for the remaining chapters in the book.

Chapter 8

This chapter discusses the architecture of the Java security package and how that architecture may be used to extend or supplant the default cryptographic algorithms that come with the SDK.

Chapter 9

This chapter discusses the APIs available to model cryptographic keys and certificates.

Chapter 10

This chapter discusses how keys can be managed within a Java program: how and where they may be stored and how they can be retrieved and validated. It also discusses programmatic transfer of digital keys.

Chapter 11

This chapter discusses message digests: how to create them, how to use them, and how to implement them.

Chapter 12

This chapter discusses how to create, use, and implement digital signatures. It also contains a discussion of signed classes.

Chapter 13

This chapter discusses the encryption available within Java Cryptography Extension (JCE), which allows developers to encrypt and decrypt arbitrary data.

Chapter 14

This chapter discusses how the Java Secure Sockets Extension (JSSE) provides SSL encryption, which can be used to encrypt data over TCP sockets. It also provides an implementation of the HTTPS Internet protocol.

Chapter 15

This chapter discusses how the Java Authentication and Authorization Service (JAAS) enables applications to authenticate users and grant them particular permissions based on their login ID or other credentials.

Appendix A

This appendix provides an annotated listing of the *java.security* file, which is the standard configuration file for the Java security architecture.

Appendix B

This appendix discusses how to keep up−to−date with information about Java's security implementation, including a discussion of Java security bugs and general resources for further information.

Appendix C

Key management in Java 1.1 was radically different than the systems we explored in the main text. This appendix discusses how key management was handled in Java 1.1; it uses classes that are still present (but deprecated) in Java 2.

Appendix D

This appendix details how the security manager operated in Java 1.1 (in the absence of an access controller) and shows how you can write an application that uses the 1.1 security model. Although most of the techniques in this appendix have been superseded in Java 2, there are exceptional cases in Java 2 when you might want to follow the tips given in this appendix.

Appendix E

In the text, we discuss how to implement standard security providers. JCE security providers require some additional steps that are outlined in this appendix.

Appendix F

This appendix is a simple reference guide to the classes discussed in this book.

What's New in This Edition

The second edition of this book provides new chapters on JSSE and JAAS, which have only recently been released. It provides updated information on JCE version 1.2.1, including modified code examples.

The remainder of the text has been reorganized, including a new chapter that presents an overview of the default sandbox and how it is administered. Information and examples are now arranged by topic rather than by package: the Diffie−Hellman key exchange algorithm, for instance, is presented in the chapters on key management rather than appearing in a chapter on JCE. We treat the core Java security packages and the three optional security packages as an integrated API (which is how it is scheduled to be packaged in the next release of Java).

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc.

101 Morris Street
Sebastopol, CA 95472
(800) 998−9938 (in the United States or Canada)
(707) 829−0515 (international/local)
(707) 829−0104 (fax)

We have a web page for this book, where we list errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/javasec2/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com/

Acknowledgments

I am grateful to the many people who have helped me with this book along the way; this book is as much a reflection of their support as anything else. I offer my heartfelt thanks to Mike Loukides and Deb Cameron for stewarding me through the editorial process.

Various drafts of this book were foisted upon my colleagues Mark Bordas, Charles Francois, David Plotkin, Nick Jacobs, and Henry Wong; I am indebted to each of them for their feedback and support, and to Lynne Doherty for all her support. In addition, I was extremely fortunate to receive technical assistance from a highly talented group of individuals: to Jim Farley, Li Gong, Jon Meyer, Michael Norman, and especially to David Hopwood, I offer my deepest thanks for all your input. Roland Schemers expertly handled my last−minute barrage of questions with patience and insight. The second edition benefited greatly from the input of Eric Brower, Jonathan Knudsen, and John Zukowski.

The staff at O'Reilly & Associates was enormously helpful in producing this book, including Colleen Gorman, the Production Editor; Robert Romano, who created the figures; Seth Maislin and Ellen Troutman−Zaig, who wrote the index; Hanna Dyer, the cover designer; David Futato, the interior designer; Mike Sierra for Tools support; and Linley Dolby for quality control.

Finally, I must offer my thanks to James for all his patience and support and for putting up with my continual state of distraction during phases of this process.

Feedback for the Author

I welcome any comments on the text you might have; despite the contributions of the people I've listed, any errors or omissions in the text are my responsibility. Please send notice of these errors or any other feedback to scott.oaks@sun.com.

Chapter 1. Java Application Security

When Java was first released by Sun Microsystems, it attracted the attention of programmers throughout the world. These developers were attracted to Java for different reasons: some were drawn to Java because of its cross−platform capabilities, some because of its ease of programming (especially compared to object−oriented languages like C++), some because of its robustness and memory management, some because of Java's security, and some for still other reasons.

Just as different developers came to Java with different expectations, so too did they bring different expectations as to what was meant by the ubiquitous phrase "Java is secure." Security means different things to different people, and many developers who had certain expectations about the word "security" were surprised to find that their expectations were not necessarily shared by the designers of Java.

This book discusses the features of Java that make it secure. In this book, we'll discuss why Java is said to be secure, what that security means (and doesn't mean), and −− most importantly −− how to use the security features of the Java platform within your own programs. This last point is actually the focus of this book: while some of Java's security features are automatically a part of all Java programs, many of them are not. In this book, we'll learn about all those features and how to utilize them in our own Java applications.

1.1 What Is Security?

The first thing we must do is to discuss just what Java's security goals are. The term "security" is vague unless it is discussed in some context; different expectations of the term "security" might lead us to expect that Java programs would be:

Safe from malevolent programs
: Programs should not be allowed to harm a user's computing environment, such as Trojan horses and harmful programs that replicate, like computer viruses.

Non−intrusive
: Programs should be prevented from discovering private information on the host computer or the host computer's network.

Authenticated
: The identity of parties involved in the program −− both the author and the user of the program −− should be verified.

Encrypted
: Data that the program sends and receives −− over the network or through a persistent store such as a filesystem or database −− should be encrypted.

Audited
: Potentially sensitive operations should always be logged.

Well−defined
: A well−defined security specification should be followed.

Verified
: Rules of operation should be set and verified.

Well−behaved
: Programs should be prevented from consuming too many system resources: too much CPU time, too much memory, and so on.

C2 or B1 certified
: Programs should have certification from the U.S. government that certain security procedures are followed.

In fact, while all of these features could be part of a secure system, only the first two were within the province of Java's 1.0 default security model. Other items in the list have been introduced in later versions of Java: authentication was added in 1.1, encryption is available as an extension to the Java 2 platform, and auditing can be added to any Java program by providing an auditing security manager. Still others of these items will be added in the future. But the basic premise remains that Java security was originally and fundamentally designed to protect the information on a computer from being accessed or modified (including a modification that would introduce a virus) while still allowing the Java program to run on that computer.

The point driving this notion of security is the new distribution model for Java programs. One of the driving forces behind Java, of course, is its ability to download programs over a network and run those programs on another machine. This is something most computer users do today within the context of a Java−enabled browser, although the idea behind portable code like this is beginning to seep into other applications, such as those based on Jini technology. Coupled with the widespread growth of Internet use −− and the public−access nature of the Internet −− Java's ability to bring programs to a user on an as−needed, just−in−time basis has been a strong reason for its rapid deployment and acceptance.

The nature of the Internet created a new and largely unprecedented requirement for programs to be free of viruses and Trojan horses. Computer users had always been used to purchasing shrink−wrapped software. Many soon began downloading software via FTP or other means and then running that software on their machines. But widespread downloading also led to a pervasive problem of malevolent attributes both in free and (ironically) in commercial software, a problem which continues unabated. The introduction of Java into this equation had the potential to multiply this problem by orders of magnitude, as computer users now download programs automatically and frequently.

For Java to succeed, it needed to circumvent the virus/Trojan horse problems that plagued other models of software distribution. Hence, the early work on Java focused on just that issue: Java programs are considered safe because they cannot install, run, or propagate viruses and because the program itself cannot perform any action that is harmful to the user's computing environment. And in this context, safety means security. This is not to say that the other issues in the above list are not important −− each has its place and its importance (in fact, we'll spend a great deal of time in this book on the third and fourth topics in that list). But the issues of protecting information and preventing viruses were considered most important; hence, features to provide that level of security were the first to be adopted. Like all parts of Java, its security model is evolving (and has evolved through its various releases); many of the notions about security in our list will eventually make their way into Java.

One of the primary goals of this book, then, is to explain Java's security model and its evolution with each subsequent release. In the final analysis, whether or not Java is secure is a subjective judgment that individual users will have to make based on their own requirements. If all you want from Java is freedom from viruses, any release of Java should meet your needs. If you need to introduce authentication or encryption into your program, you'll need to use a 1.1 or later release of Java. If you have a requirement that all operations be audited, you'll need to build that auditing into your applications. If you really need conformance with a U.S. government−approved definition of security, Java is not the platform for you. We take a very pragmatic view of security in this book: the issue is not whether a system that lacks a particular feature qualifies as "secure" according to someone's definition of security. The issue is whether Java possesses the features that meet your needs.

1.2 Software Used in This Book

The information in this book is based on the Java 2 Standard Edition, version 1.3 (or 1.3, for short). There are slight differences between how Java security operates in 1.2 (that is, the Java 2 Standard Edition, version 1.2) and 1.3. When we refer to a specific release, we'll use its number; otherwise, we'll say Java 2 to refer to either platform.

In addition, there are great differences in how Java security operates between the Java 1.1 and the Java 2 platform. While we concentrate on Java 1.3, the end of each chapter contains a section that elucidates the differences between Java 1.3 and previous releases of Java. Some of the very different topics of Java 1.1 are presented in the appendices of this book; it is not generally recommended that you use the facilities and APIs discussed there since they are not compatible with the Java 2 platform.

We present information in this book from three standard Java extensions: the Java Cryptography Extension (JCE) version 1.2.1, the Java Secure Sockets Extension (JSSE) version 1.0.2, and the Java Authentication and Authorization Service (JAAS) version 1.0. Each of these contributes certain technologies to the Java security story. These extensions require 1.3.

Information about the extensions is presented thoughout the book as it makes sense. The JSSE API defines a set of classes that are used to perform SSL operations, and these are discussed in a separate chapter. The JSSE API also defines a set of classes that are used for key management; these are discussed along with the classes in the core API that handle key management. So even though these three packages are standard extensions, we recommend that you install them now along with the SDK so that you can become familiar with their features when they arise. In version 1.4, all these extensions are scheduled to be included in the core SDK, which is another reason why it helps to think of them as an integrated unit.

In the next few pages, we'll discuss how to obtain and install the platform and extensions. Configuring the extensions may require some steps that you don't understand right now because they have various security options that apply to them. However, we recommend that you just follow the instructions for now and install the extensions. The extensions use Java's standard security framework, and as we discuss each aspect of the framework, we detail how the extensions relate to that aspect. Thus, while the core features of each extension is discussed in its own chapter, information about the extensions appears throughout the book.

1.2.1 The Java 2 Platform

The core Java 2 platform supplies the basic facilities of Java security:

- A configurable security policy that lets you prevent Java programs from reading your files, making network connections to other hosts, accessing your printer without permission, and so on. This policy is based on Java's access controller, which in turn depends upon Java's class loaders, security manager, and language protections.
- The ability to generate message digests if you want a simple (but not secure) way to determine if data your program reads has been altered.
- The ability to generate digital signatures if you want to detect if data your program reads has been altered (or if you want to send data and enable the recipient of that data to detect if the data was altered in transit).
- A key management system to manage the keys necessary for digital signatures.
- An extensible infrastructure to support all of this.

Java 2 version 1.3 can be obtained for Solaris, Linux, and Windows systems from http://java.sun.com/j2se/1.3/. If you need Java for other platforms, check with your platform vendor or check http://java.sun.com/cgi−bin/java−ports.cgi.

The Java 2 platform contains two flavors: the Software Development Kit (SDK, also known historically as the JDK) and the Java Runtime Environment (JRE). Administration of the security model applies to both the JRE and SDK, but to use the security APIs that we discuss, you'll need the SDK (which includes the JRE). Throughout this book, we'll use the environment variable *$JDKHOME* to refer to the directory in which the Java 2 SDK was installed and the *$JREHOME* variable to refer to the directory in which the Java 2 JRE was installed. If you installed the SDK into *C:\files\jdk1.3* then *$JDKHOME* would be *C:\files\jdk1.3* and *$JREHOME* would be *C:\files\jdk1.3\jre*.

Installed or Bundled Extensions?

When you work with the extensions that we use in this book, you have the option of treating them as installed or bundled extensions.

Installed extensions are much easier to work with: they require no special configuration once they are installed. However, they must be installed into special directories within *$JREHOME*, and they may require files in *$JREHOME* to be modified. Depending on your setup, this may require special operating system privileges.

A bundled extension requires no special installation privileges, but it does require you to set up things within your environment: you must modify your classpath, and you must set up special policy files. In addition, some of this configuration must be done programatically, so this option will not work for third−party applications. We assume in our examples that you've set up the extensions as installed extensions.

1.2.2 The Java Cryptography Extension

JCE leverages the Java 2 core platform's security architecture to provide a variety of cryptographic operations:

- Encryption (Ciphers)
- Secure Key Exchange
- Secure Message Digests
- An alternate key management system

JCE can be downloaded from http://java.sun.com/products/jce/. Version 1.2.1 is an important version because it takes advantage of a change in the policy of the United States regarding export controls of cryptographic engines. Prior to early 2000, the United States government considered cryptographic engines to be a munition and severely restricted the export of such technology. After this policy was changed in early 2000, JCE 1.2.1 was modified to meet the new standards. As a result, although it performs strong encryption, JCE 1.2.1 can be exported from the United States.

JCE consists of some documentation and a *lib* directory that contains four jar files: *US_export_policy.jar, jce1_2_1.jar, local_policy.jar,* and *sunjce_provider.jar*. Like most extensions, you can install JCE as a bundled or unbundled extension.

To use JCE as an installed extension, you must:

- Copy the four jar files to *$JREHOME/lib/ext*
- Add the following line to *$JREHOME/lib/security/java.security*:

```
security.provider.3=com.sun.crypto.provider.SunJCE
```

This line should immediately follow the line that reads:

```
security.provider.2=com.sun.rsajca.Provider
```

To use JCE as an unbundled extension, you must:

- Add the four jar files to your classpath.
- Add some configuration information to *$HOME/.java.policy*. The information to be added depends on where you have placed the jar files; if you've put JCE into */files/jce1.2.1* then the appropriate lines are:

  ```
  grant codebase "file:///files/jce1.2.1/lib/US_export_policy.jar" {
      permission java.security.AllPermission;
  };
  grant codebase "file:///files/jce1.2.1/lib/jce1_2_1.jar" {
      permission java.security.AllPermission;
  };
  grant codebase "file:///files/jce1.2.1/lib/local_policy.jar" {
      permission java.security.AllPermission;
  };
  grant codebase "file:///files/jce1.2.1/lib/sunjce_provider.jar" {
      permission java.security.AllPermission;
  };
  ```

  You must substitute the appropriate path for */files/jce1.2.1*. Note that this is a URL; you use forward slashes no matter what your platform. On Microsoft Windows, the beginning of the appropriate URL is *file:/C:/files/jce1.2.1*.
- In every program that you run, you must insert the following line:

  ```
  Security.addProvider(new com.sun.crypto.provider.SunJCE(  ));
  ```

More details about how this works can be found in later chapters. Chapter 8, discusses the addition to the *java.security* file and its programmatic alternative, and the *.java.policy* file is discussed in Chapter 2.

1.2.3 The Java Secure Sockets Extension

JSSE provides Secure Sockets Layer (SSL) encryption facilities. If you need to communicate with an SSL server or SSL client, you can use the APIs in this extension. If you are writing both a client and server and want to do encryption, you can use this extension or you can use the cipher facilities of JCE.

JSSE can be downloaded from http://java.sun.com/products/jsse/. Version 1.0.2 takes advantage of the relaxed export restrictions of the U.S. and is exportable. Unlike JCE, however, there are still two different versions of JSSE: one for domestic use (use within the United States and Canada) and one for global use. The difference between these two versions is that the domestic version allows you to substitute new implementations of the SSL algorithms. Such substitution is still prohibited by export rules, so the global version does not allow it. However, both versions provide exactly the same API and the same key strength for their encryption.

JSSE consists of documentation and a *lib* directory containing three jar files: *jcert.jar, jnet.jar*, and *jsse.jar*. To use JSSE as an installed extension, you must:

- Copy the three jar files to *$JREHOME/lib/ext*.
- Add the following line to *$JREHOME/lib/security/java.security*:

  ```
  security.provider.4=com.sun.net.ssl.internal.ssl.Provider
  ```

This line should immediately follow the line you inserted for JCE.

To use JSSE as an unbundled extension, you must:

- Add the three jar files to your classpath.
- In every program that you run, you must insert the following line:

```
Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider(  ));
```

1.2.4 The Java Authentication and Authorization Service

JAAS provides for user authentication within the Java platform. It performs a unique function in the Java platform. All of the core facilities of Java's security design are intended to protect end users from the influences of developers: end users give permissions to developers to access resources on the end user's machine. JAAS, on the other hand, allows developers to grant (or deny) access to their programs based on the authentication credentials provided by the user.

JAAS can be downloaded from http://java.sun.com/products/jaas/. It comes in two parts: a Java library which defines the interface to the service (the JAAS proper), and platform−specific modules to perform the authorization (the JAAS modules). Sample modules are available to perform authentication based on JNDI directory services, Windows NT login services, and Solaris login services.

JAAS itself contains documentation and a *lib* directory with a single jar file (*jaas.jar*). The jar file should either be installed into *$JREHOME/lib/ext*, or the user must add it to her classpath.

The *lib* directory of the JAAS modules contains an additional jar file (*jaasmod.jar*) that must be handled similarly. It also contains platform−specific shared libraries. On Solaris systems, these libraries must be installed into *$JREHOME/lib/sparc*. If that is not possible, the libraries can be placed into any directory (e.g., */files/jaasmod1_0/lib*) and that directory can be added to the user's LD_LIBRARY_PATH.

On Microsoft Windows systems, these libraries are named *nt.dll*, *nt.lib*, and *nt.exp* and they must be installed into *$JREHOME\bin*. If that is not possible, then you must set the `java.library.path` property on the command line. For instance, if the libraries are in *\files\jaasmod1_0\lib*, you would specify the following property on the command line:

```
−Djava.library.path=\files\jaasmod1_0\lib
```

No modification to the *java.security* file is required for JAAS.

1.2.5 More About Export Controls

The U.S. is not the only government that regulates the use of encryption, and encryption software can face import restrictions as well as export restrictions. In France, for example, it is illegal to import many encryption packages without a license. Other countries have regulations for cryptography, but in most cases they are less onerous than those of the United States. However, it is always wise to check your local policies to be sure (see Appendix B, for resources to find more information about these limitations).

Even though the U.S. has relaxed its export rules, some restrictions still apply. You may not export either JCE or JSSE (and, hence, any programs that use them) to the following countries: Afghanistan, Cuba, Iran, Iraq, Libya, North Korea, Serbia/Montenegro (Yugoslavia), Sudan, Syria and parties listed on the Denied and Restricted Parties List (available at http://bxa.fedworld.gov/prohib.html). Additionally, it is Sun company policy not to ship products to Burma.

The encryption extensions, like many aspects of the Java platform, allow for third−party implementations; just like you can buy a third−party JDBC driver, you can buy third−party implementations of JCE. However, many of the popular algorithms that are used by the extensions are patented algorithms, which also restricts their use. RSA Data Security, Inc. holds a patent in the U.S. on several algorithms involving RSA encryption and digital signatures; Ascom System AG in Switzerland holds both U.S. and European patents on the IDEA method of performing encryption. If you live in a country where these patents apply, you can't use these underlying algorithms without paying a licensing fee to the patent holder. In particular, this means that many of the third−party security providers and third−party implementations of JCE cannot be used within the United States because of patents held by RSA (although some of them have reached a licensing agreement with RSA Data Security, Inc. −− again, it is best to check with the provider to see what restrictions might apply). Sun has an agreement with RSA Data Security to redistribute its implementation of the RSA algorithms.

Encryption and Weaponry

The whole question of importing and exporting encryption technology occurs because it is often classified as a munition. While this position is sometimes questioned, it comes from a long tradition in computer science.

During WWII, the Allies waged a successful and pivotal campaign in the Atlantic against the Axis navy. The success of this campaign was greatly due to the work of Alan Turing, who with his colleagues broke the German encryption algorithm known as Enigma. Turing was also one of the founding fathers of modern computer science, much of which was based on the work he developed in service to his country during the war.

Ironically, the reward that Turing reaped for his efforts was that some years after the war, he was arrested and forced to undergo harmful chemical treatments because he was gay. There's an odd parallel here: many of the harsh restrictions that are sometimes placed on encryption technology make no more sense in a world with a global Internet than did England's persecution of Alan Turing in the 1950s. Perhaps the relaxation of export restrictions is a good sign in general.

Note that import and export restrictions apply only to the encryption technology contained within JCE and JSSE. Although the core Java APIs perform important cryptographic operations, those operations are not considered to be munition−grade operations.

1.2.6 Other Software Versions

Though Java−enabled browsers are very popular, we do not discuss most of the popular ones. This is because their security implementations are very different from the official Java security model. Both Netscape and Microsoft, for example, introduced new (and proprietary) APIs to allow for security extensions. Both companies also developed their own (again proprietary) mechanism to sign applets.

The better way to run Java applets from within a browser is to use the Java Plug−in, which comes standard with every release of the Java 2 platform; when you install Java 2, you install the Java Plug−in. In Netscape 6 and later releases, the Java Plug−in is the only way to run Java applets; there is no Java virtual machine built into Netscape 6. The Plug−in is also compatible with Internet Explorer 4.x and higher, as well as Netscape 4.x versions.

The security model of the Java Plug−in is exactly the same as that of the Java platform that we describe within this book. Where the security model of older browsers is different, we point that out, and we provide directions to the vendor's web sites that give information on their non−standard systems.

1.3 The Java Sandbox

When Java security is discussed, the discussion typically centers around Java's applet−based security model −− the security model that is embodied by Java−enabled browsers. It's considered "applet−based" because in early versions of Java, it applied only to applets that run within a Java−enabled browser. In the Java 2 platform, however, this security model can apply to any Java application as well as to the Java Plug−in, which allows newer browsers to run Java 2 applets. The Java 2 security model is also configurable by an end user or system administrator so that it can be made less restrictive than earlier implementations of that model.

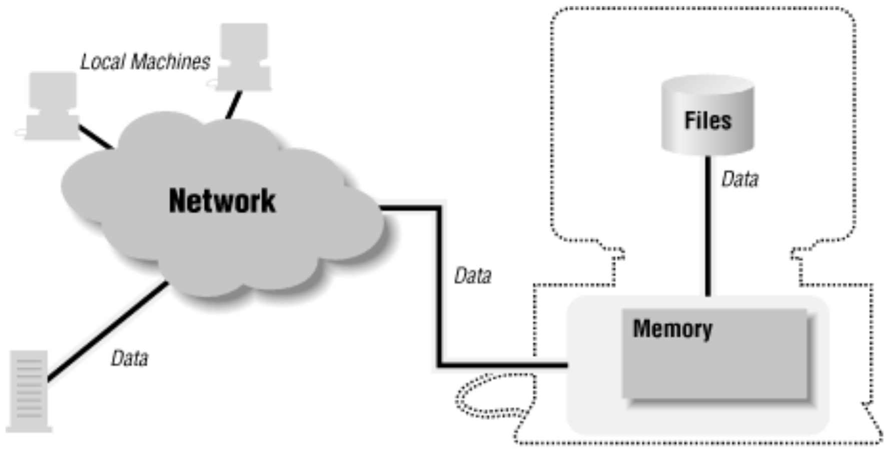
This security model centers around the idea of a sandbox. The idea is when you allow a program to be hosted on your computer, you want to provide an environment where the program can play (i.e., run), but you want to confine the program's play area in certain bounds. You may decide to give the program certain toys to play with (i.e., you may decide to let it have access to certain system resources), but in general, make sure that the program is confined to its sandbox.

This analogy works better when you consider it from the view of a close relative rather than from the view of a parent. If you're a parent, you probably consider the purpose of a sandbox to be to provide a safe environment for your child to play in. When my niece Rachel visits me, however, I consider the purpose of a sandbox not (only) to be to protect her, but also to protect my grandmother's china *from* her. I love my niece, but I can't give her leave to run through my house; I enjoy running the latest cool applet on the Internet, but I can't give it leave to run through my filesystem.

The Java sandbox is responsible for protecting a number of resources, and it does so at a number of levels. Consider the resources of a typical machine as shown in Figure 1−1. The user's machine has access to many things:

- Internally, it has access to its local memory (the computer's RAM).
- Externally, it has access to its filesystem and to other machines on the local network.
- For running applets, it also has access to a web server, which may be on its local (private) net or may be on the Internet.
- Data flows through this entire model, from the user's machine through the network and (possibly) to disk.

Figure 1−1. A machine has access to many resources



Each of these resources needs to be protected, and those protections form the basis of Java's security model. We can imagine a number of different−sized sandboxes in which a Java program might run:

- A sandbox in which the program has access to the CPU, the screen, keyboard, and mouse, and to its own memory. This is the minimal sandbox −− it contains just enough resources for a program to run.
- A sandbox in which the program has access to the CPU and its own memory as well as access to the web server from which it was loaded. This is often thought of as the default state for the sandbox.
- A sandbox in which the program has access to the CPU, its memory, its web server, and to a set of program−specific resources (local files, local machines, etc.). A word−processing program, for example, might have access to the *docs* directory on the local filesystem, but not to any other files.
- An open sandbox, in which the program has access to whatever resources the host machine normally has access to.

The sandbox, then, is not a one−size−fits−all model. Expanding the boundaries of the sandbox is always based on the notion of trust: when my one−year−old niece comes to visit, there's very little in the sandbox for her to play with, but when my six−year−old godchild comes to visit, I trust that I might give her more things to play with. In the hands of some visitors, a toy with small removable parts would be dangerous, but when I trust the recipient, it's perfectly reasonable to include that item in the sandbox. And so it is with Java programs: in some cases, I might trust them to access my filesystem; in other cases, I might trust them to access only part of my filesystem; and in still other cases, I might not trust them to access my filesystem at all.

1.3.1 Applets, Applications, and Programs

In early versions of Java, only applets were run within a sandbox. In the Java 2 platform, all programs have the potential to run in a sandbox. Applets that run through the Java Plug−in or the appletviewer will always run in a sandbox, and applications that are run via the command line (or by clicking an icon on the desktop) may optionally be set up to use a sandbox. Applications also have the option of programatically installing new versions of the sandbox.

Hence, in the Java 2 platform there is little distinction between the security level of an applet and an application. There are programmatic differences, of course, but both are subject to the same security model, and the security model for both is administered and programmed in the same way. There is one significant difference, however: applets always run with Java's security model (even if that model has been administered such that the applet is allowed to do anything it wants to), and an application will only run under the security model if it is told to do so. This is typically done by the end user by specifying a command−line parameter; it may be done by the program developer who specifies that parameter in a script that starts the application, and it may be done by the developer who inserts code into his program.

Any program, including an applet, can change the behavior of the sandbox under certain circumstances. However, most of them will use Java's default sandbox. The default sandbox, as we'll see in Chapter 2, is very flexible; it allows the user who runs the program to determine exactly how the sandbox will operate. This moves the definition of the security policy to the end user or system administrator of the machine running the program.

1.3.2 Anatomy of a Java Program

The anatomy of a typical Java program is shown in Figure 1−2. Each of the features of the Java platform that appears in a rectangle plays a role in the development of the Java security model. In particular, the elements of the Java security policy are defined by:

The bytecode verifier
: The bytecode verifier ensures that Java class files follow the rules of the Java language. In terms of resources, the bytecode verifier helps enforce memory protections for all Java programs. As the figure implies, not all classes are subject to bytecode verification.

The class loader

One or more class loaders load all Java classes. Programatically, the class loader can set permissions for each class it loads.

The access controller

The access controller allows (or prevents) most access from the core API to the operating system, based upon policies set by the end user or system administrator.

The security manager

The security manager is the primary interface between the core API and the operating system; it has the ultimate responsibility for allowing or preventing access to all system resources. However, it exists mostly for historical reasons; it defers its actions to the access controller.

The security package

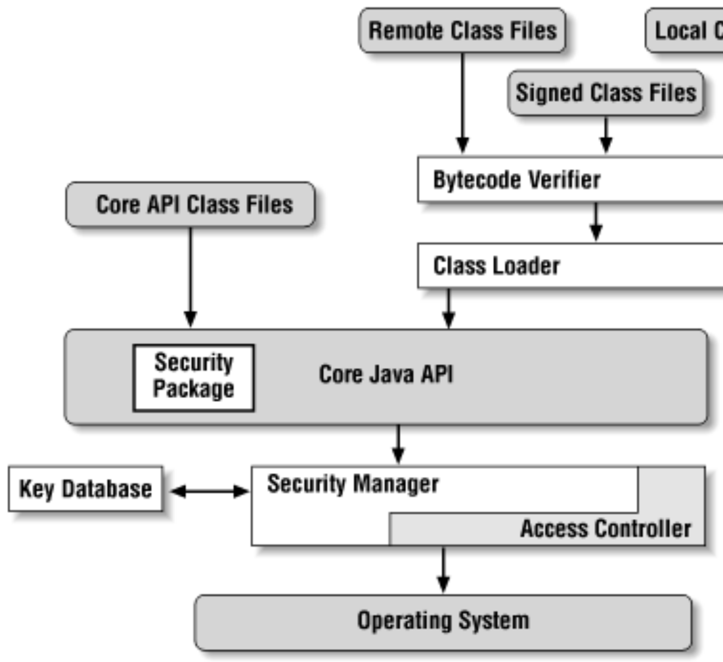
The security package (that is, classes in the `java.security` package as well as those in the security extensions) allows you to add security features to your own application as well as providing the basis by which Java classes may be signed. Although it is only a small box in this diagram, the security package is a complex API and discussion of it is broken into several chapters of this book. This includes discussions of:

◊ The security provider interface −− the means by which different security implementations may be plugged into the security package
◊ Message digests
◊ Keys and certificates
◊ Digital signatures
◊ Encryption (through JCE and JSSE)
◊ Authentication (through JAAS)

The key database

The key database is a set of keys used by the security infrastructure to create or verify digital signatures. In the Java architecture, it is part of the security package, though it may be manifested as an external file or database.

Figure 1−2. Anatomy of a Java application



The last two items in this list have broad applicability beyond expanding the Java sandbox. With respect to the sandbox, digital signatures play an important role because they provide authentication of who actually provided the Java class. As we'll see, this provides the ability for end users and system administrators to grant very specific privileges to individual classes or signers. But a digital signature might be used for other applications. Let's say that you're deploying a payroll application throughout a large corporation. When an employee sends a request to view his payroll information, you really want to make sure that the request came from that employee rather than from someone else in the corporation. Often, this type of application is secured by a simple password, but a more secure system could require a digitally signed request before it sent out the payroll information.

This list is also a rough outline of the path we'll take through this book. We'll start by looking at the default sandbox and how it can be administered. Following that, we'll look at the details of everything that makes up that sandbox, from the bytecode verifier through the access controller. Then we'll move into the security APIs that allow you to add digital signatures and encryption to your own applications.

1.4 Security Debugging

The Java security packages include debugging code that you can enable via a system property. The property in question is `java.security.debug`, and it may be set to the following values:

all

Turn on all the debugging options.

access

Trace all calls to the `checkPermission( )` method of the access controller. This allows you to see which permissions your code is requesting, which calls are succeeding, and which ones are failing.

This option has the following sub−options. If no sub−option is specified, then all are in force:

stack
: Dump the stack every time a permission is checked.

failure
: Dump the stack only when a permission is denied.

domain
: Dump the protection domain in force when a protection is checked.

jar

When processing a signed jar file, print the signatures in the file, their certificates, and the classes to which they apply.

policy

Print information about policy files as they are parsed, including their location in the filesystem, the permissions they grant, and the certificates they use for signed code.

scl

Print information about the permissions granted directly by a secure class loader (rather than granted through a policy file).

These options should be given as a comma−separated list (including the sub−options for the access option). For example, to see the permissions granted by the secure class loader and see a stack trace when a permission check fails, you would specify `−Djava.security.debug=scl,access,failure` on the command line.

JSSE extends this facility by consulting the `javax.net.debug` property for the following options:

all

Turn on all options and sub−options.

ssl

Turn on SSL debugging. This option has the following sub−options (all of which are in force if none are specified):

record
: Print a trace of each SSL record (at the SSL protocol level).

handshake
: Print each handshake message as it is received.

keygen
: Print key generation data for the secret key exchange.

session
: Print SSL session activity.

defaultctx
: Print the default SSL initialization information.

sslctx
: Print information about the SSL context.

sessioncache
: Print information about the SSL session cache.

keymanager	Print information about calls to the key manager.
trustmanager	Print information about calls to the trust manager.
data	For handshake tracing, print out a hex dump of each message.
verbose	For handshake tracing, print out verbose information.
plaintext	For record tracing, print out a hex dump of the record.

As you progress through the samples in the book, you can turn various options on in order to see more information about what's going on.

1.5 Summary

Security is a multifaceted feature of the Java platform. There are a number of facilities within Java that allow you to write a Java application that implements a particular security policy, and this book will focus on each of those facilities in turn. These features are important within a Java−enabled browser, and they are also important with Java applications, particularly as applications become more distributed.

In addition, the security package allows us to create applications that use generic security features −− such as digital signatures −− for many purposes aside from expanding the Java sandbox. This other use of the security package will also be a constant theme throughout this book.

Chapter 2. The Default Sandbox

In this chapter, we're going to explore the default sandbox that is used by most Java programs. The default sandbox is designed to allow an end user or system administrator to easily change the parameters of the sandbox so that certain programs may run with a very specific set of permissions. If you're interested in how an applet running in the Java Plug−in can read a file, this chapter provides the information that you need. If you're interested in having your own applications use the (possibly modified) sandbox, this is the place to be.

The information in this chapter is targeted to end users and system administrators: those are the people who are ultimately responsible for defining the security policies that their programs use. Except in special circumstances, it is not possible to change security policies programmatically: in the default sandbox, there is no API that a developer can use that allows her to change a security policy. If you want your program to read a local file, then you must tell the user who will run that program to modify the security policy of his machine *before* he runs your program. However, developers do need to understand the concepts (and especially the terms) that we define in this chapter.

In the next few chapters, we'll discuss the programmatic details of how the sandbox is implemented; this will give you a better understanding of how Java security works and allow you to develop your own programs that implement a different security policy. But we'll start with the default sandbox, an administratable, flexible model used by most Java programs.

2.1 Elements of the Java Sandbox

From an administrative point of view, the sandbox is composed of five elements:

Permissions

A permission is a specific action that code is allowed to perform. Permissions may be specific (e.g., the file *C:\WINDOWS\Desktop\My Documents\Chapter2.fm* can be read but not written or deleted) or very general (e.g., the code can do anything it wants).

Permissions are composed of three elements: the *type* of the permission, its *name*, and its *actions*. The type of the permission is required; it is the name of a particular Java class that implements the permission. Although no programming is involved in administering the default sandbox, administrators must know the Java class name of various permissions in order to allow code to perform those operations.

A few permissions (like `java.security.AllPermission`, which allows code to do anything) require no name. Otherwise, the name is based on the type of the permission; for example, the name of a file permission is a file or directory name. The names of permissions are often specified as wildcards, such as all files in a directory or all hosts on the local network.

The actions of a permission also vary based on the type of the permission; many permissions have no action at all. The action specifies what may be done to the target; a file permission may specify that a particular file can be read, written, deleted, or some combination of those actions.

Here are three examples of permissions. The first carries only a type; the second carries a type and a name; the third carries a type, a name, and a set of actions:

```
permission java.security.AllPermission;
permission java.lang.RuntimePermission "stopThread";
permission java.io.FilePermission "/tmp/foo", "read";
```

Code sources

Code sources are the location from which a class has been loaded along with information about who signed the class, if applicable. The location is specified as a URL, which follows standard Java practice: code can be loaded from the file system through a file−based URL or from the network via a network−based URL.

If code is signed, information about the signer is included in the code source. However, it's important to note that the URL and signer information in a code source are both optional. Classes can be assigned permissions based only on the URL from which the class was loaded, based only upon who signed the class, or a combination of both. Hence, it is not required that code be signed in order for it to carry special permissions. The URL within a code source is called a codebase.

Protection domains

A protection domain is an association of permissions with a particular code source. Protection domains are the basic concept of the default sandbox; they tell us things like code loaded from *www.oreilly.com* is allowed to read files on my disk, code loaded from *www.sun.com* is allowed to initiate print jobs, or code that is loaded from *www.klflote.com* and signed by Scott is allowed to do anything it wants.

Policy files

Policy files are the administrative element that controls the sandbox. A policy file contains one or more entries that define a protection domain; less formally, we can say that an entry in a policy file grants specific permissions to code that is loaded from a particular location and/or signed by a particular entity.

Programs vary in the way in which they define policy files, but there are usually two policy files in use: a global policy file that all instances of the virtual machine use and a user−specific policy file. Policy files are simple files that can be created and modified with a text editor, and the JRE comes with a tool (`policytool`) that allows them to be administered as well.

Keystores

Code signing is one way in which code can be granted more latitude. The rationale behind code signing is discussed in Chapter 7, but if you are assured that code you are running came from an organization that you trust, you may feel comfortable allowing that code to read the files on your disk, send jobs to the printer, or whatever.

Signed code depends on public key certificates, and there is a lot of administration that takes place when you use certificates. The certificates themselves are held in a location (usually a file) called the keystore. If you are a developer, the keystore is consulted to find the certificate used to sign your code; if you are an end user or system administrator, the keystore is consulted when you run signed code to see who actually signed the code.

In the next few sections, we'll look at these elements in more depth.

2.2 Permissions

Every Java class carries a set of permissions that defines the activities that the class is allowed to perform. The parameters of the sandbox are wholly defined by these permissions. When a Java program attempts to perform a sensitive operation, the permissions for all active classes are consulted: if every class carries the permission to perform the operation, then the operation is permitted to continue. Otherwise, an exception is thrown in the code, and the operation fails.

Which Permissions Apply

In Chapter 5, we'll explain in detail what we mean when we say that the permissions of "active" classes are used to determine if an operation succeeds. In general, though, it matters which code initiates an operation. An applet cannot initiate a print job unless it has been explicitly granted permission to do so. However, users can print applets by initiating the print job through the browser. Even though the applet code is active, the browser code is still granted permission to execute the print operation.

Classes that make up the core Java API are always given permission to perform any action. All other classes, including those on the classpath, must explicitly be given permission to perform sensitive operations. For the most part, these permissions are listed in various policy files, along with the code source to which they apply. End users and system administrators define the parameters of the sandbox by administering these policy files.

The permissions within the virtual machine are based on an extensible system of Java classes. Hence, any Java API or application can define its own permissions and ensure that the user has granted those permissions to the API before it can run. For example, the Jini API defines a `net.jini.discovery.DiscoveryPermission` class. Classes that want to participate in a Jini community must be granted this permission in order to execute the code that participates in Jini's discovery protocol. In general, then, you may need to become familiar with arbitrary permission types in order to use certain APIs.

Here are the standard permissions used by the core Java API.

File Permissions

Type

`java.io.FilePermission`

Name

The name of the file that you want to operate on. Of course, filenames are platform−specific. Hence, */myclasses/xyz* is a valid name for a file permission on a Unix system, but not on a Macintosh (where an equivalent name might be *System Disk:myclasses:xyz*). Because a backslash has special meaning to the code that parses a policy file, to specify a Microsoft Windows file you must use two backslashes: *C:\\myclasses\\xyz*.

It's generally better to use a special syntax that specifies platform−independent names. The string *${/}* is expanded at runtime to be the file separator symbol for the platform so that the string *${/}files${/}jre1.3${/}lib* can be used to specify the same directory hierarchy on any platform. In addition, you can use any standard Java property when you specify a filename so that the string *${user.home}${/}.keystore* will expand to */home/sdo/.keystore* or *C:\WINDOWS\ .keystore* or whatever other platform−specific name is appropriate.

Two wildcard patterns are available for specifying filenames: an asterisk matches all files in a given directory and a hyphen matches all files that reside in a directory hierarchy. Hence the string *${user.home}${/}**

matches all files that are in the user's home directory, but no files that are in directories contained within the user's home directory. The string *${user.home}${/}−* matches any file in the user's home directory and all of its subdirectories. If you want to match all files on a particular machine, you may specify the special token `<<ALL FILES>>`.

Actions

read, write, delete, and execute. You can specify multiple actions for a particular file by separating the actions with commas.

Examples

```
// Allow all types of access to all files
permission java.io.FilePermission "<<ALL FILES>>",
                                  "read,write,delete,execute";
// Allow the user's home directory to be read
permission java.io.FilePermission "${user.home}/−", "read";
```

Remember that file permissions can be overridden by platform permissions: simply because you specify that the Java program has permission to read the */etc/shadow* file (a privileged Unix file), a Java application will be unable to do so unless it is run by the Unix super user. Code is always able to read files from the directory hierarchy from which it was loaded; this particular permission cannot be changed administratively, although it can be changed programatically.

Property Expansion and the Policy Class

The file separator expansion we have mentioned is a specific example of a general rule: names in permissions may contain any defined environment variable .

This property substitution allows us to use one set of configuration files no matter what the underlying platform since we can use standard Java properties to hide those platform−specific details. This is particularly important when specifying filenames for file permissions in a policy file.

If the `policy.expandProperties` property in the *java.security* file is set to `false`, however, substitution will not occur and these strings should not be used. If they are used, they will be treated as literal strings and fail.

Socket Permissions

Type

`java.net.SocketPermission`

Name

`hostname:port`, where each component of the name may be specified by a wildcard. In particular, the hostname may be given as a hostname (possibly DNS qualified) or an IP address. The leftmost position of the hostname may be specified as an asterisk, such that the host *piccolo.East.Sun.COM* would be matched by each of these strings:

```
piccolo
piccolo.East.Sun.COM
*.Sun.COM
*
129.151.119.8
```

The port component of the name can be specified as a single port number or as a range of port numbers (e.g., 1−1024). When a range is specified, either side of the range may be excluded:

```
1024 (port 1024)
1024− (all ports greater than or equal to 1024)
−1024 (all ports less than or equal to 1024)
1−1024 (all ports between 1 and 1024, inclusive)
```

Actions

accept, listen, connect, and resolve. These map into the normal socket usage: accept is used to see if the program can accept an incoming connection from a particular host; listen is used to see if the program can accept any incoming socket connections; connect is used to see if the program can make a connection to a particular host; and resolve is used to see if the IP address for a particular hostname can be obtained from the machine's name service.

Code is always able to make a socket connection to (and accept a connection from) the host from which it was loaded, even if that specific permission is not listed in a policy file. That particular permission cannot be changed by an end user or administrator (though it can be changed by a developer).

Examples

```
// Allow all socket operations
permission java.net.SocketPermission "*:1−",
                            "accept,listen,connect,resolve";
// Allow outgoing connections to oreilly.com
permission java.net.SocketPermission "*.oreilly.com:1−",
                            "connect,resolve";
```

Property Permissions

Type

`java.util.PropertyPermission`

Name

The name of the Java virtual machine property that you want to access. Property permission names are specified as dot−separated names (just as they are in a Java property file); in addition, the last element can be a wildcard asterisk: `*`, `a.*`, `a.b.*`, and so on. Note, however, that no other element can be a wildcard; `*.b.c` does not allow you to access the `a.b.c` property.

Actions

read and write.

Examples

```
// Read standard java properties
permission java.util.PropertyPermission "java.*", "read";
// Create properties in the sdo package
permission java.util.PropertyPermission "sdo.*", "read,write";
```

Runtime Permissions

Type

java.lang.RuntimePermission

Name

Various. All names in this class are dot−separated names and are subject to the same wildcard asterisk matching as the property permission class. Here is a list of the names of the runtime permissions used by the core Java API:

accessClassInPackage.<name>
: Allow code to access classes in the named package.

accessDeclaredMembers
: Allow code to use reflection to access the `private` or `protected` members of other classes.

createClassLoader
: Allow code to instantiate a class loader.

createSecurityManager
: Allow code to instantiate a security manager, which programmatically controls the sandbox.

defineClassInPackage.<name>
: Allow code to define classes in the named package.

exitVM
: Allow code to shut down the entire virtual machine.

getClassLoader
: Allow code to retrieve the class loader for a particular class.

getProtectionDomain
: Allow code to retrieve a protection domain object for a particular class.

loadlibrary.<name>
: Allow code to load the library with the given name.

modifyThread
: Allow code to change certain thread parameters.

modifyThreadGroup
: Allow code to change certain thread group parameters.

queuePrintJob
: Allow code to initiate a print job.

readFileDescriptor
: Allow code to read file descriptors (certain files opened by code in other protection domains).

setContextClassLoader
: Allow code to set the context class loader for a thread.

setFactory
: Allow code to create socket factories.

setIO
: Allow code to redirect the `System.in`, `System.out`, or `System.err` streams.

setSecurityManager
: Allow code to set the security manager in use.

stopThread
: Allow code to call the `stop( )` method of the thread class.

writeFileDescriptor
: Allow code to write file descriptors (certain files opened by code in other protection domains).

Many of these permissions have unusual rules. The `accessClassInPackage` permission is used to test only those packages that are listed in the `package.access` property defined within the *$JREHOME/lib/security/java.security* file. By default, that is set to classes within the `sun` package. If you wanted additionally to prevent code from accessing classes in the `sdo` package, you would edit the *java.security* file so that it defined a property like this:

```
package.access=sun.,sdo.
```

If you then want to permit access to some sub−packages, you could list the following permission:

```
permission java.lang.RuntimePermission
           "accessClassInPackage.sdo.foo";
```

Then code would be able to access classes in the `sdo.foo` package but not other classes in any other `sdo` package.

The `defineClassInPackage` permission behaves similarly, except that the operation in question is whether or not the code is able to load classes in the given packages. The property used in that test is named `package.definition`, which by default is not set. Both the define and access permissions also require cooperation of the class loader; class loaders determine whether or not to enforce those permissions. The default class loaders enforce the access permissions but not the define permissions: you may define a class in any package, even if you set the `package.definition` property to attempt to prevent it. This permission is useful only with a custom class loader, as we'll examine in Chapter 6.

Note, however, that you cannot define any class in the `java` package, regardless of any permission that you have set.

The `exitVM` permission is automatically granted by many environments, including the default Java command line (but not including the appletviewer or Java Plug−in).

The `stopThread` permission is listed in the default global policy file so that any code can stop a thread. This is for backward compatibility and may change in future releases (the `stop( )` method, of course, has been deprecated).

The `modifyThread` and `modifyThreadGroup` permissions are used only if the thread or thread group the program is attempting to modify is the system thread group. Programs generally create and manipulate threads in the main thread group, a subgroup of the system thread group, so these permissions rarely come into play.

Actions

None. You either have permission to perform a runtime operation, or you do not.

Examples

```
// Allow code to initiate printing
permission java.lang.RuntimePermission "queuePrintJob";
```

AWT Permissions

Type

`java.awt.AWTPermission`

Name

There are six names supported by this class, as listed here:

accessClipboard
: Allow access to the system's global clipboard.

accessEventQueue
: Allow direct access to the event queue.

createRobot
: Allow code to create the AWT Robot class.

listenToAllAWTEvents
: Allow code to listen directly to the event dispatcher.

readDisplayPixels
: Allow AWT robots to read the pixels of the display.

showWindowWithoutWarningBanner
Allow window frames to be created without an identifying warning banner.

The `accessEventQueue` permission is a little unusual because of the way in which event queues are handled by most implementations of the virtual machine. Most implementations, including the appletviewer and the Java Plug−in, set up a separate event queue for each applet codebase. This means that an applet can use the methods of the event queue class to access its own event queue; the `accessEventQueue` permission does not need to be granted in that case. However, if you install the sandbox via a command−line argument, then this does not apply, and your code will need this permission in order to call methods of the event queue class.

All applications still need the `listenToAllAWTEvents` permission in order to register an event listener with the default toolkit.

Actions

None.

Examples

```
// Allow code to use the robot class fully
permission java.awt.AWTPermission "createRobot";
permission java.awt.AWTPermission "readDisplayPixels";
```

Net Permissions

Type

java.net.NetPermission

Name

There are three names associated with this class. `specifyStreamHandler` allows new stream handlers to be installed into the URL class. HTTP authentication (which is not actually implemented by default Java implementations) requires two permissions: `setDefaultAuthenticator` to install the authentication class and `requestPasswordAuthentication` to complete the authentication.

Actions

None.

Examples

```
// Allow a stream handler to be installed
permission java.net.NetPermission "specifyStreamHandler";
```

Security Permissions

Type

`java.security.SecurityPermission`

Name

Various. Security permission names are subject to wildcard asterisk matching and include all the valid strings that can be passed to the `checkSecurityAccess( )` method of the security manager. All possible names are listed in here; as we explore the security API throughout the rest of the book, we'll mention when the names apply.

addIdentityCertificate
: Add a certificate to an `Identity` object.

clearProviderProperties.<provider name>
: Remove all properties from the named provider.

createAccessControlContext
: Allow creation of an access controller context.

getDomainCombiner
: Allow collapsing of protection domains.

getPolicy
: Retrieve the class that implements the sandbox policy.

getProperty.<prop name>
: Read the given security property.

getSignerPrivateKey
: Get the private key from a `Signer` object.

insertProvider.<provider name>
: Add the named provider to the set of security providers.

loadProviderProperties.<provider name>
: Bulk load the properties of the named provider.

printIdentity
: Print out the contents of the `Identity` class.

putAllProviderProperties.<provider name>
: Bulk update the properties of the named provider.

putProviderProperty.<provider name>
: Add a property to the named security provider.

removeIdentityCertificate

Remove the certificate of an `Identity` object.

removeProvider.<provider name>
Remove the named provider from the set of security providers.

removeProviderProperty.<provider name>
Remove a property from the named security provider.

setIdentityInfo
Set the information string of an `Identity` object.

setIdentityPublicKey
Set the public key of an `Identity` object.

setPolicy
Set the class that implements the sandbox policy.

setProperty.<prop name>
Set the given security property.

setSignerKeyPair
Set the key pair within a `Signer` object.

setSystemScope
Set the system−wide `IdentityScope`.

Many of these names are not used in the Java 2 platform because they refer to deprecated classes.

Actions

None.

Serializable Permissions

Type

`java.io.SerializablePermission`

Names

`enableSubstitution` and `enableSubclassImplementation`. The first of these permissions allows the `enableResolveObject( )` method of the `ObjectInputStream` and the `enableReplaceObject( )` method of the `ObjectOutputStream` classes to function. The latter permission allows the `ObjectInputStream` and `ObjectOutputStream` classes to be subclassed, which would potentially override the `readObject( )` and `writeObject( )` methods.

Actions

None.

Reflection Permissions

Type

`java.lang.reflect.ReflectPermission`

Name

`suppressAccessChecks` . If granted, this permission allows reflection to examine the `private` variables of arbitrary classes.

Actions

None.

All Permissions

Type

`java.security.AllPermission`

Name

None.

Actions

None. This class represents permission to perform any operation −− including file, socket, and other operations that have their own permission classes. Granting this type of permission is obviously somewhat dangerous; this permission is usually given only to classes within the Java API and to classes in Java extensions.

2.3 Keystores

Java code can be signed, which entails obtaining digital certificates and running the `jarsigner` (or equivalent) tool. You can grant permissions to code that is signed by a particular entity.

If you choose to handle signed code, you must establish a keystore to hold the public keys of the signing entity. Before you run the signed code, you must obtain the public key certificate of the signing entity and install that certificate into your keystore. Some browsers (e.g., Netscape 6) allow you to accept the public key certificate when you first run the signed program, but usually you must install the public key certificate before running the program.

Administration of the keystore is handled by the `keytool` utility (see Chapter 10). By default, the keystore is held in a file called *.keystore* in the user's home directory. When you install a public key certificate into the keystore, you give that certificate an alias that is used to look up the certificate in the future. For example, my public key certificate lists my full name and other identifying information, but you may enter it into your keystore with an alias of sdo. This alias is the information that you list in a policy file.

2.4 Code Sources

Code sources are a combination of codebases and signers. The signer field must match the alias listed in the keystore, as we've just described. Codebases can be any valid URL. Because they are URLs, codebases always use forward slashes, even if the code is being read from the filesystem: *file:/C:/files/jdk1.3/* is the correct specification for the directory *C:\files\jdk1.3\* (although remember that if the drive is the default drive, you can leave it out and specify the URL as *file:///files/jdk1.3/*).

Codebases can use property substitution much like we showed when discussing file properties. Hence, the codebase for the Java extension directory can be universally specified as *file:${java.home}/lib/ext/*. This trick applies somewhat to the classpath: if you want a particular set of permissions to apply to all classes on the classpath then you can use the codebase *file:${java.class.path}/*. However, this works only when there is a single directory or JAR file in the classpath since only those cases result in a correct URL.

The ending of the codebase URL is very important. There are four cases:

- The URL specifies a jar file (*http://www.sun.com/sdo/sdoapp.jar*).

  Only the classes in the jar file belong to the codebase.
- The URL ends with a slash (*http://www.sun.com/sdo/*).

  Only class files in the given directory belong to the codebase. Jar files in the given directory do not belong to the codebase.
- The URL ends with an asterisk (*http://www.sun.com/sdo/**).

  Both jar files and class files in the given directory belong to the codebase. However, jar and class files in subdirectories of the URL do not belong to the codebase.
- The URL ends with a hyphen (*http://www.sun.com/sdo/−*).

  All jar and class files in the given directory and its subdirectories belong to the codebase.

Note this directory syntax is not affected by package names. If you want to give permissions to the class `com.sdo.PayrollApp`, you list the directory in which the `com` subdirectory appears (e.g., *http://www.sdo.net/*, not *http://www.sdo.net/com/sdo/*).

2.5 Policy Files

In order to administer the Java sandbox, you list the various permissions we've discussed in a Java policy file. Java virtual machines can use any number of policy files, but there are two that are used by default. There is a global policy file named *$JREHOME/lib/security/java.policy* that is used by all instances of a virtual machine on a host. We consider this to be a global policy file because it allows an environment where the JRE is mounted from a common server by several machines; each of these machines will share the definitions in this file.

In addition, there is a user−specific policy file called *.java.policy* that may exist in each user's home directory (*$HOME* on UNIX systems, *C:\WINDOWS* on single−user Windows 98 systems, and so on). The set of permissions given to a program is the union of permissions contained in the global and user−specific policy files.

Policy files are simple text files. You can administer them with `policytool`, or you can edit them by hand. Hand editing is discouraged (in 1.3, `policytool` writes a warning at the top of the file not to edit it by hand), but real programmers still edit them by hand. Policy files are also used with JAAS, in which case their syntax changes slightly and you must edit them by hand (at least until 1.4, when JAAS becomes integrated with the SDK). So first, we'll see how they look, and then we'll look at how they are created with `policytool`.

Here's how a typical policy file might look:

```
keystore "${user.home}${/}.keystore";

// Grant these permissions to code loaded from O'Reilly, regardless of
// whether the code is signed.
grant codeBase "http://www.oreilly.com/" {
  permission java.io.FilePermission "/tmp", "read";
  permission java.lang.RuntimePermission "queuePrintJob";
};

// Grant these permissions to code loaded from Sun but only if it is
// signed by sdo.
grant signedBy "sdo", codeBase "http://www.sun.com/" {
  permission java.security.AllPermission;
};

// Grant these permissions to code signed by jra, no matter where it
// was loaded from
grant signedBy "jra" {
  permission java.net.SocketPermission "*:1024−",
                      "accept, connect, listen, resolve";
};

// Grant these permissions to any code, no matter where it came
// from or whether it is signed
grant {
    permission java.util.PropertyPermission
                      "java.version", "read";
};
```

Note how the policy file combines all the elements of the sandbox: the code sources (the combination of `signedBy` and `codeBase` elements) are associated with various permissions to create protection domains; the entire file is subject to the given keystore.

The first line of this example tells the virtual machine to consult the keystore in the file *$HOME/.keystore* when it needs to check the certificates of entities that have signed code. The next four blocks of text define protection domains: code that is loaded from O'Reilly's web site has permission to read files in */tmp* and to start a print job; code that is signed by sdo and loaded from Sun's web site has permission to do anything it wants to; code that is signed by jra is able to operate on any nonprivileged socket; and all code is allowed to read the `java.vendor` system property. In each of these blocks, the syntax is the same: the word grant is followed by a code source and then a set of permissions enclosed by braces. The code source is composed of a codebase and a signer, either of which may be blank.

In most cases, code that you run is loaded from only a single location: if you run the applet contained on the *java.sun.com* home page, you load all the code from *java.sun.com*. So only the permissions in a protection domain with that code source are in force while you run that code. However, in certain circumstances you may be running code loaded from multiple locations, particularly when RMI calls are involved: you may download an applet from *www.mystockinfo.com* that downloads RMI stubs to talk to a server at *www.freestockquotes.com* and so on. In that case, the permissions that are in force are the intersection of the permissions associated with the *www.mystockinfo.com* codebase and the *www.freestockquotes.com* codebase. In order for the applet to be able to talk to the stock quote server, both codebases must contain the appropriate socket permissions.[1]

[1] There are subtle variations of this scheme discussed in Chapter 5.

If you leave out the codebase or the signer from a code source in this file, then that entry is considered a wildcard. In the example above, code that is signed by jra can open sockets regardless of where the code was loaded from; if code signed by jra is loaded from the O'Reilly web site, then that code can open sockets, read files in */tmp*, and initiate print jobs. If the signer is not present, then it does not matter whether or not the code is signed: both signed code and unsigned code loaded from the O'Reilly web site can read files in */tmp* or initiate print jobs. Similarly, if both the codebase and signer are omitted, then the permissions listed apply to code loaded from any location, whether or not it is signed.

In sum, the permissions granted to a particular code source are taken as the union of all permissions between all policy files used by the virtual machine for all relevant codebases and/or signers. The permissions given to an application are determined by the intersection of all permissions in all active code sources.

2.5.1 The policytool

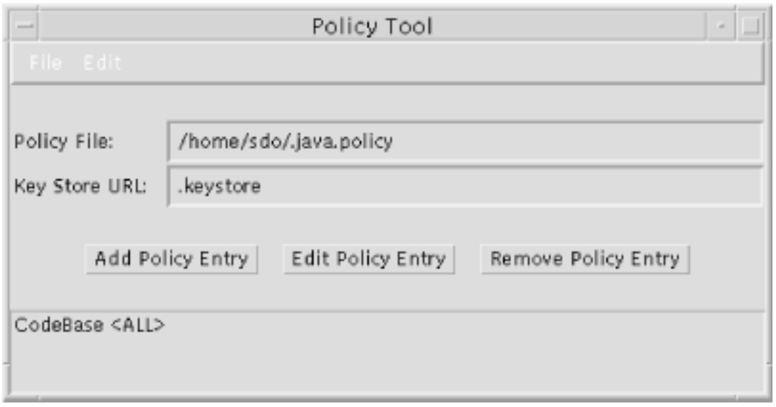
The `policytool` allows you to manage entries in a *java.policy* file. Unlike the other tools that we'll discuss in this book, `policytool` is a graphical tool. It takes one command−line argument: `−file filename`, which allows you to specify the name of the initial policy file to edit. This defaults to *$HOME/.java.policy*; if that file does not exist, then no file is loaded by default.

When you first start `policytool`, you see a blank window with two pull−down menus: File and Edit. The File menu allows you to create a new policy file, open a different policy file, and save (or save as) the current policy file. Keep in mind that `policytool` is designed to edit a single file at a time; you must save one file before starting work on another. The File menu also allows you to view a warning log, which will contain information about parsing errors when reading an existing file or errors in reading the keystore.

2.5.1.1 Managing policy codebases

The initial screen for this tool displays the name of the currently loaded policy file (which is blank if no file has been loaded), the name of the keystore referenced within this file, buttons to add, edit, or remove policy entries, and a list of the current set of policy entries. In this context, a policy entry is the URL from which classes will be loaded, that is, a codebase or a code source. Hence, a single policy entry may contain many individual permissions. In Figure 2−1 we've loaded the default *java.policy* file, which has one policy entry: an entry that grants permissions to all codebases.

Figure 2−1. policytool loaded with one policy entry



Policy File: /home/sdo/.java.policy
Key Store URL: .keystore

CodeBase <ALL>

Note that the keystore entry for this file is *.keystore*. You can change that value through an option under the Edit menu.

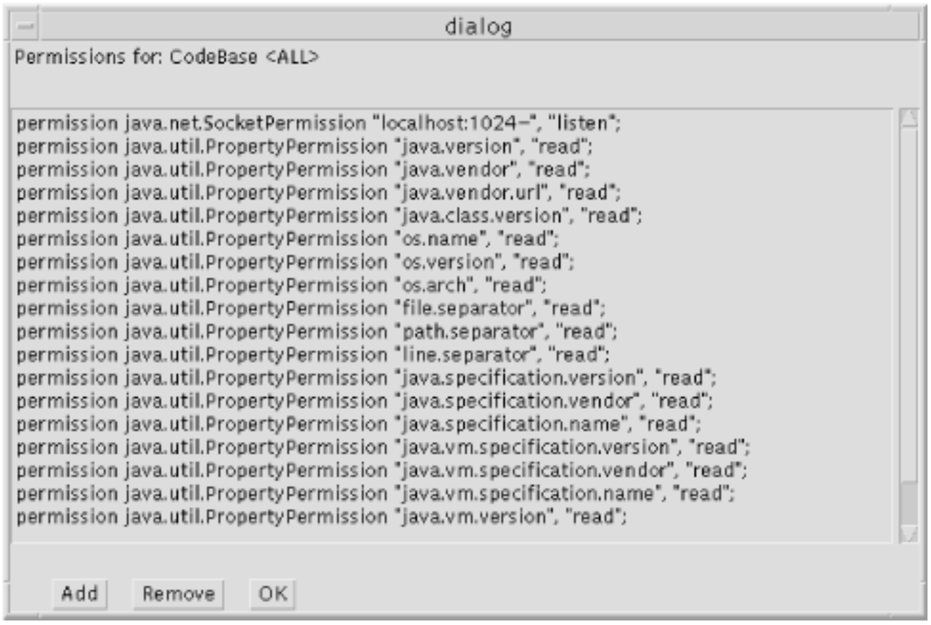
You can add new codebases to this file by selecting the Add Policy Entry button; when you add a policy entry, you are allowed to specify a URL and a signer (both of which are optional). The entry for the signer should be an alias in the keystore; if you enter a signer who is not in the keystore, you'll get a warning, but the operation will continue.

You may delete codebases by selecting one and pressing the Remove Policy Entry button. Selecting a codebase and pressing the Edit Policy Entry button allows you to edit the specific set of permissions for a codebase.

2.5.1.2 Managing permissions

When you press the Edit Policy Entry button, you get a window similar to that shown in Figure 2−2. This window lists all permissions that are associated with the given codebase and provides the opportunity to add or remove individual permissions.

Figure 2−2. A set of permissions for a codebase



Permissions for: CodeBase <ALL>

```
permission java.net.SocketPermission "localhost:1024-", "listen";
permission java.util.PropertyPermission "java.version", "read";
permission java.util.PropertyPermission "java.vendor", "read";
permission java.util.PropertyPermission "java.vendor.url", "read";
permission java.util.PropertyPermission "java.class.version", "read";
permission java.util.PropertyPermission "os.name", "read";
permission java.util.PropertyPermission "os.version", "read";
permission java.util.PropertyPermission "os.arch", "read";
permission java.util.PropertyPermission "file.separator", "read";
permission java.util.PropertyPermission "path.separator", "read";
permission java.util.PropertyPermission "line.separator", "read";
permission java.util.PropertyPermission "java.specification.version", "read";
permission java.util.PropertyPermission "java.specification.vendor", "read";
permission java.util.PropertyPermission "java.specification.name", "read";
permission java.util.PropertyPermission "java.vm.specification.version", "read";
permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
permission java.util.PropertyPermission "java.vm.specification.name", "read";
permission java.util.PropertyPermission "java.vm.version", "read";
```

The window where you add permissions contains three pulldown menus in which you can select the permission type, name, and action(s). The list of permission types in the pulldown is limited to those defined in the core API; if you need to add a permission in an extension or a third−party package, you must enter the type explicitly. When you select a type, the pulldown for the permission target name changes to match valid names for the type. However, since many types accept arbitrary names, you often need to type the target name explicitly into the box; for example, when you select a socket permission, you must type the hostname and/or port name explicitly. Even though `policytool` provides an administrative GUI for policy files, you still must be very familiar with the names of permissions as well as the targets and actions they accept.

2.5.2 Permissions Outside of Policy Files

Virtually all of the permissions granted to code comes from policy files. However, advanced applications are allowed to grant additional permissions to code that they load, and standard Java class loaders grant some additional permissions to every class that they load.

Classes that are loaded from the filesystem are always granted permission to read files in the directory hierarchy from which they were loaded. Classes that are loaded via HTTP are always granted permission to establish a connection back to the host from which they were loaded; they are also granted permission to accept a connection from that host.

2.6 The Default Sandbox

And now, putting this all together, let's examine the default sandbox of various Java environments:

Java applications

For applications invoked via the Java command line, the sandbox is initially disabled. To enable the sandbox, you must specify the `java.security.manager` property like this:

```
piccolo% java −Djava.security.manager <other args>
```

Applications may also enable the sandbox programatically by installing a security manager, as we discuss in Chapter 4.

Once enabled, the security manager will use the two default policy files to determine the parameters of the sandbox. You can specify an additional policy file to be used with the `java.security.policy` property:

```
piccolo% java −Djava.security.policy=<URL>
```

You can specify a full URL (e.g., with an `http:` or `file:` protocol) or simply list a filename. If you want the given policy file to be the only policy file used (bypassing the ones in *$JREHOME/lib/security* and the user's home directory), specify two equals signs:

```
piccolo% java −Djava.security.policy==<URL>
```

Putting this all together, here's how we would run the class `PayrollApp` in the default sandbox with additional permissions loaded from the file *java.policy* in the local directory:

```
piccolo% java −Djava.security.manager \
              −Djava.security.policy=java.policy PayrollApp
```

Appletviewer

The appletviewer installs a security manager programatically; it cannot be disabled. It will use the standard policy files; to use additional policy files, specify the appropriate policy argument with the `−J` argument:

```
piccolo% appletviewer −J−Djava.security.policy=<URL>
```

Although it obeys the default rules for accessing classes in packages (via the `accessClassInPackage` permission discussed earlier), the appletviewer also allows you to restrict or allow access to classes in the `sun` package through a special property file. That property is set in the appletviewer properties menu.

The Java Plug−in

The Java Plug−in installs a security manager programatically; it cannot be disabled. It will use the standard policy files; to use additional policy files, you must use the Java Plug−in Control Panel. On the advanced tab of that panel, you can specify the desired `java.security.policy` argument.

The Java Plug−in supports an alternate sandbox. This sandbox is used whenever the Plug−in runs an applet that has been signed. When the Plug−in encounters a signed jar file, it will present a dialog box to the user. The user has the option of giving the signed code permission to perform any operation for this session only or anytime it runs code signed by the given organization. Otherwise, the code will run with normal permissions (based on its codebase and the permissions in the relevant policy files).

This is the model used by Netscape 6 as well (since Netscape 6 uses the Java Plug−in for all applets).

Other Java−enabled browsers

Older versions of Netscape and all versions of Internet Explorer define their own sandbox. Those sandboxes are completely unrelated to the policy−based model we've discussed here. They provide the same restrictions that we have discussed in this chapter (applets cannot read files, can only open sockets back to the host from which they were loaded, have limited property permissions, and no other permissions); it's simply that you cannot modify the sandbox administratively.

These browsers do allow code to be signed, in which case the user can optionally grant the code permission to perform many operations. In addition, they allow developers to ask for particular operations to be performed using proprietary APIs. Check with the browser vendor for more details.

2.6.1 The Default Policy File

One way or another, the three common Java environments will use the same policy files to determine the parameters of the sandbox. By default, users do not have a *.java.policy* file in their home directory, which means that the default set of permissions for all Java programs running in the sandbox is defined by the *$JREHOME/lib/security/java.policy* file.

Here are the contents of that file in 1.3:

```
// Installed extensions are granted all permissions
grant codeBase "file:${java.home}/lib/ext/*" {
        permission java.security.AllPermission;
};

// All code is allowed the following permissions
grant {
    // Anyone can call Thread.stop(  ), though only for
    // backward compatibility.
    permission java.lang.RuntimePermission "stopThread";

    // allows anyone to listen on unprivileged ports
    permission java.net.SocketPermission "localhost:1024-", "listen";

    // "standard" properies that can be read by anyone
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version",
                                              "read";
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator", "read";
    permission java.util.PropertyPermission "path.separator", "read";
    permission java.util.PropertyPermission "line.separator", "read";

    permission java.util.PropertyPermission
                        "java.specification.version", "read";
    permission java.util.PropertyPermission
                        "java.specification.vendor", "read";
    permission java.util.PropertyPermission
                        "java.specification.name", "read";
    permission java.util.PropertyPermission
                        "java.vm.specification.version", "read";
    permission java.util.PropertyPermission
                        "java.vm.specification.vendor", "read";
    permission java.util.PropertyPermission
                        "java.vm.specification.name", "read";
    permission java.util.PropertyPermission "java.vm.version", "read";
    permission java.util.PropertyPermission "java.vm.vendor", "read";
    permission java.util.PropertyPermission "java.vm.name", "read";
};
```

By default, installed extensions can perform any operation. All other code is allowed to call `Thread.stop( )`, to listen on an unprivileged port, and to read a limited set of system properties. And that's it: no file access, no other socket access, and so on (other than the file and socket access granted to all files that we mentioned earlier). Even the ability to listen on an unprivileged port is of dubious value: in order to complete a socket connection, the program is required to accept a connection from a particular IP address. The accept permission is not granted in this file; it must be enabled elsewhere in order for a program to open a server socket.

2.7 The java.security File

Many parameters of the default sandbox are controlled by entries in the *java.security* file. This file (*$JREHOME/lib/security/java.security*) can be edited by system administrators, which is particularly effective when it is shared by several end users (though end users, of course, can administer it themselves if required).

That file has several entries that we'll examine throughout this book; an annotated version of it appears in Appendix A. In terms of the default sandbox, here are the important entries:

```
policy.expandProperties=true
policy.allowSystemProperty=true
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

When we say that there are two default policy files, it's because of the entries in this file (note that the Java property `java.home` expands to what we've been calling JREHOME). If you would prefer a different set of policy files to be used by default, you can edit the *java.security* file and change the URLs that are used. You may specify any number of URLs, but they must be numbered consecutively beginning with 1.

As we've seen, users can specify any number of policy files on the command line as well. To prevent users from specifying additional policy files, set the `allowSystemProperty` property to `false`. A site that has that value set to false, removes the user's *.java.policy* entry from this file, and makes the *java.security* file uneditable by end users has established a sandbox that cannot be modified by end users.

We've shown several examples of how properties can be used in policy files in order to make them more portable. If you want to disable substitution of properties, set the `expandProperties` property to `false`.

2.8 Comparison with Previous Releases

The default sandbox is essentially unchanged between 1.2 and 1.3. In 1.2, there is no prohibition against defining classes in the `java` package.

In Java 1.1, the default sandbox is very different. The 1.1 sandbox is determined solely by the security manager installed by the Java program; although it is possible to write a security manager that allows end users and administrators to configure different security policies, few programs followed that course. For most Java applications, this meant that no security manager was ever installed, and the program ran with complete permissions. Java applets run through the appletviewer and early, 1.1−based versions of the Java Plug−in are subject to strict, nonconfigurable restrictions. Using the signing tool of 1.1 (`javakey`), it is possible to sign Java applets; these applets can then be given permission to perform any operation. However, the 1.1−based signing infrastructure has been deprecated, and an applet signed with `javakey` will not be given any special permissions in Java 2.

If you need to understand the 1.1 default sandbox, see Chapter 4 for a discussion of the security manager. Appendix D, shows how you can build a 1.1 application to run other Java applications with a security policy that you've written yourself.

2.9 Summary

In this chapter, we examined the bounds and adaptability of the Java sandbox. The default sandbox is set up to be administered through a series of policy files, which contain sets of explicit permissions associated with code; this association depends on where the code was loaded from and/or who signed the code.

In the next few chapters, we'll delve into the details of the sandbox and how it is implemented. We'll learn how the sandbox is built up from the low−level components of the Java platform. If you're a developer, we'll learn how to interact with those components while preserving the sandbox model or how you can replace it altogether. And if you're just interested in security, you'll learn the details of how Java achieves its goal of platform security.

Chapter 3. Java Language Security

The first components of the Java sandbox that we will examine are those built into the Java language itself. These components primarily protect memory resources on the user's machine, although they have some benefit to the Java API as well. Hence, they are primarily concerned with guaranteeing the integrity of the memory of the machine that is hosting a program: in a nutshell, the security features within the Java language want to ensure that a program will be unable to discern or modify sensitive information that may reside in the memory of a user's machine. In terms of applets, these protections also mean that applets will be unable to determine information about each other; each applet is given, in essence, its own memory space in which to operate.

In this chapter, we'll look at the features of the Java language that provide this type of security. We'll also look at how these features are enforced, including a look at Java's bytecode verifier. With a few exceptions, the information in this chapter is largely informational; because the features we are going to discuss are immutable within the Java language, there are fewer programming considerations than we'll find in later chapters. However, the information presented here is crucial in understanding the entire Java security story; it is very helpful in ensuring that your Java environment is secure and in assessing the security risks that Java deployment might pose. The security of the Java environment is dependent on the security of each of its pieces, and the Java language forms the first fundamental piece of that security.

As we discuss the language features in this chapter, keep in mind that we're only dealing with the Java language itself −− following the common thread of this book, not all security features we're going to discuss apply when the language in question is not Java. If you use Java's native interface to run arbitrary C code, that C code will be able to do pretty much anything it wants to do, even when it violates the precepts outlined in this chapter.

3.1 Java Language Security Constructs

In this chapter, we're concerned primarily with how Java operates on things that are in memory on a particular machine. Within a Java program, every entity −− that is, every object reference and every primitive data element −− has an access level associated with it. To review, this access level may be:

private
: The entity can only be accessed by code that is contained within the class that defines the entity.

Default (or package)
: The entity can be accessed by code that is contained within the class that defines the entity, or by a class that is contained in the same package as the class that defines the entity.

protected
: The entity can only be accessed by code that is contained within the class that defines the entity, by classes within the same package as the defining class, or by a subclass of the defining class.

public
: The entity can be accessed by code in any class.

The notion of assigning data entities an access level is certainly not exclusive to Java; it's a hallmark of many object−oriented languages. Since the Java language borrows heavily from C++, it's not surprising that it would borrow the basic notion of these access levels from C++ as well (although there are slight differences between the meanings of these access modifiers in Java and in C++).

As a result of this borrowing, the use of these access modifiers is generally thought of in terms of the advantage such modifiers bring to program design: one of the hallmarks of object−oriented design is that it permits data hiding and data encapsulation. This encapsulation ensures that objects may only be operated upon through the interface the object provides to the world, instead of being operated upon by directly manipulating the object's data elements. These and other design−related advantages are indeed important in order to develop large, robust, object−oriented systems. But in Java, these advantages are only part of the story.

In a language like C++, if I create a `CreditCard` object that encapsulates my mother's maiden name and my account number, I would probably decide that those entities should be private to the object and provide the appropriate methods to operate on those entities. But nothing in C++ prevents me from cheating and accessing those entities through a variety of back−door operations. The C++ compiler is likely to complain if I write code that attempts to access a private variable of another class, but the C++ runtime isn't going to care if I convert a pointer to that class into an arbitrary memory pointer and start scanning through memory until I find a location that contains a string with 16 digits −− a possible account number. In C++ systems, no one typically worried about such occurrences because all parts of the system were presumed to originate from the same place: it's my program, and if I want to work around my data model to get access to that data, then so be it.[1]

[1] In a large project with multiple programmers, there's a strong argument that such an attitude on the part of an individual programmer is not to be dismissed so lightly, but we'll let that pass.

Things change with Java. I might be surfing to play some cool game applet on *www.EvilSite.org*, and then I might go shopping at *www.Acme.com*. When my Java wallet applet runs, I'd hate for the applet that is still running from *www.EvilSite.org* to be able to access the private `CreditCard` object that is contained in my Java wallet −− and while it's necessary for *www.Acme.com* to know that I have a valid `CreditCard` object, I don't necessarily feel comfortable telling them my mother's maiden name. Because I'm now in the midst of a dynamic system with active programs from multiple sites, I need to make sure that the data entities are accessed by only those objects that are supposed to have access to them. It's obvious that I want protection from *EvilSite.org*, whom I don't want to know about the `CreditCard` object contained in my Java wallet. But I also want to be protected from *Acme.com*, a site I feel relatively comfortable about, but who should not be granted access to all the data elements of an object that it must use.

This is only one example of why the Java platform must provide memory integrity: that is, it must ensure that entities in memory are accessed only when they are allowed to be and that these entities cannot be somehow corrupted. To that end, Java always enforces the following rules:

Access methods are strictly adhered to.

In Java, you cannot be allowed to treat a `private` entity as anything but private: the intentions of the programmer must always be respected. Object serialization involves an exception to this rule; we'll give more details about that a little bit later.

Programs cannot access arbitrary memory locations.

This is easy to ensure, as Java does not have the notion of a pointer. For example, casting between an `int` and an `Object` is strictly illegal in Java.

Entities that are declared as final must not be changed.

Final variables in Java are considered constants; they are immutable once they are initialized. Consider the havoc that could ensue if the `final` modifier were not respected:

- ◊ A `public final` variable could be changed, drastically altering the behavior of a program. If a rogue applet swapped the values of the variables `EAST` and `WEST` in the `GridBagConstraints` class, for example, any new applets would be laid out incorrectly (and probably incomprehensibly). That's a rather benign example of what could potentially be a dramatic security flaw.
- ◊ A subclass could override a `final` method, altering the behavior of a class. One of the features of the Java API is that threads are not allowed to raise their priority above a certain maximum priority (typically, the priority of the thread group to which the thread belongs). This feature is enforced by the `setPriority( )` method of the `Thread` class, which is a `final` method; allowing that method to be overridden would defeat the security mechanisms.

  This feature is used for virtually all of Java's security checks: performing an operation requires calling a `final` method in a Java class; only that `final` method can trap into the operating system to execute the operation. That `final` method is responsible for making sure the operation does not proceed if it would violate the security policy in place.
- ◊ A subclass could be created from a `final` class, with similar results. In Java, strings are considered as constants -- their value may not be changed once the string has been created. If the `String` class could be subclassed, this rule could not be enforced.

Variables may not be used before they are initialized.

If a program were able to read the value of an uninitialized variable, the effect would be the same as if it were able to read random memory locations. A Java class wishing to exploit this defect might then declare a huge uninitialized section of variables in an attempt to snoop the memory contents of the user's machine. To prevent this type of attack, all local variables in Java must be initialized before they are used, and all instance variables in Java are automatically initialized to a default value.

Array bounds must be checked on all array accesses.

Like the access modifiers that started this discussion, bounds checking is generally thought of in terms other than security: the prime benefit to bounds checking is that it leads to fewer bugs and more robust programs. But it has security benefits as well: if an array of integers happens to reside in memory next to a string (which, in memory, is an array of characters), writing past the end of the array of integers would change the value of the string. The effect of this is generally a bug, but it could be exploited as a security hole as well: if the string held the destination account number for an electronic funds transfer, we could change the destination account number by willfully writing past the end of the array of integers.[2]

[2] This type of attack is not as farfetched as it might seem; an early version of Netscape Navigator suffered from just this type of security hole. When long URLs were typed into the Goto field, the Netscape C code that read the string overwrote the bounds of the array where the characters were to be stored and clobbered a key location in memory, which allowed a security breach.

Objects cannot be arbitrarily cast into other objects.

Given the class fragment:

```
public class CreditCard {
        private String acctNo;
}
```

and the rogue class:

```
public class CreditCardSnoop {
        public String acctNo;
}
```

then the following code cannot be allowed to execute:

```
CreditCard cc = Wallet.getCreditCard(  );
CreditCardSnoop snoop = (CreditCardSnoop) cc;
System.out.println("Ha!  Your account number is " + snoop.acctNo);
```

Hence, Java does not allow arbitrary casting between objects; an object can only be cast to one of its superclasses or its subclasses (if, in the latter case, the object actually is an instance of that subclass). Note that the Java virtual machine is much stricter about this rule than the Java compiler is. In the example above, the compiler would complain about an illegal cast. We could satisfy the compiler by changing the code as follows:

```
Object cc = Wallet.getCreditCard(  );
CreditCardSnoop snoop = (CreditCardSnoop) cc;
```

Only the virtual machine will know if the returned object actually is of type `CreditCard` or not. In this case, then, the virtual machine is responsible for throwing a `ClassCastException` when the `snoop` variable is assigned to thwart the attack.

These are the techniques by which the Java language ensures that memory locations are read and written only when such access should normally be allowed. This restriction protects the user's machine from the outside: if I download an applet onto my machine, I don't want that applet accessing the private variables of my `CreditCard` class. However, if that applet has a private variable within it, nothing prevents me (depending on my operating system) from using a program outside of the browser to scan the memory on my system and figure out somehow what value that particular variable has. Similarly, nothing prevents me from having another program outside the browser change the value of a particular variable that is held in memory on my machine.

If you're an applet developer and are worried about this type of problem, you're pretty much on your own to come up with a solution to it. This might be particularly troublesome if you had, say, a variable somewhere in your applet that held a Boolean value indicating whether or not the user was licensed for a particular operation; a very clever user can go outside the browser and manipulate the machine's memory so that the integrity of your licensing scheme is violated. This problem is not new to Java, but it's not solved by Java either.

3.1.1 Object Serialization and Memory Integrity

There is one general exception to the rules about public, private, and protected access in Java. Object serialization is a feature of Java that allows an object to be written as a series of bytes; when those bytes are read someplace else, a new object is created that has the same state as the original object. Object serialization has two main purposes: it's used extensively in the RMI API to allow clients and servers to exchange objects, and it's used whenever you need to save a particular object to disk and want to recreate the object at some later point in time.

The murky issue here is just what constitutes an object's state. In the case of our `CreditCard` object, the account number is pretty basic to creating that object, but it's a variable that needs to be private for the reasons we've been discussing. In order for object serialization to work, it must have access to those private variables so it can correctly save and restore the object's state. That's why the object serialization API can access and save all private variables of an object (as well as its default, protected, and public variables). Similarly, the object serialization API is able to store those values back into the private data members when the object is actually reconstituted.

Depending on your perspective, this is a good thing or a bad thing. From a security perspective, it can be a bad thing: if the `CreditCard` object is saved to disk, something else can come along and read all that information from the disk file. Worse yet, the file could be edited in such a way that the object will be recreated in a completely different state than it originally had, with potentially damaging results.

In theory, this is the same problem we just discussed about influences outside the browser being able to read and write the private data of objects that are held in memory (which may help to explain why object serialization works this way by default). In practice, however, it's much easier to change the data in a binary file than to figure out how to access and change the value of an object in memory. Hence, object serialization has two additional mechanisms associated with it that make it more secure.

The first of these is that object serialization can only occur on objects that implement the `java.io.Serializable` interface (or its subclass, the `java.io.Externalizable` interface). The `Serializable` interface requires no methods, so it can be thought of simply as a flag to the virtual machine that says, "Hey, virtual machine −− I've thought about the security aspects of this class, and it's okay if you serialize it by writing out all its data." By default, an object is not serializable, lest its internal private state be violated.

The second of these mechanisms is that object serialization respects the `transient` keyword associated with a variable: if our account number in the `CreditCard` class were declared as `private transient`, then object serialization would not be allowed to read or write that particular variable. This lets us design classes that can be stored and reconstituted without showing their private data to the world.

Of course, a `CreditCard` object without an account number is worthless; what we really need is something that can save and reconstitute the transient data in such a way that the data can't be compromised. This is achieved by having our class implement the `writeObject( )` and `readObject( )` methods. The `writeObject()` method is responsible for writing out all data in the class; it typically uses the `defaultWriteObject( )` method to write out all nontransient data, and then it writes the transient data out in any format it desires. Similarly, the `readObject( )` method uses the `defaultReadObject( )` method to read the data and then must restore the corresponding transient data. It's your decision how to save and reconstitute the transient data so that its integrity is preserved, but this will mean that you'll want to use one of the encryption APIs we discuss in Chapter 13.

Storing and reconstituting the transient data can also be achieved by implementing the `Externalizable` interface and implementing the `writeExternal( )` and the `readExternal( )` methods of that interface. The difference in this case is that these two methods are now responsible for saving and reconstituting the entire state of the object −− no data can be stored or reconstituted by any default methods.

Using either of these techniques, you have the ability to protect any sensitive data contained in your objects, even if you choose to share those objects over the network or save those objects to some sort of persistent storage.

3.2 Enforcement of the Java Language Rules

The list of rules we outlined above are fine in theory, but they must be enforced somehow. We've always been taught that overwriting the end of an array in C code is a bad thing, but I somehow still manage to do it accidentally all the time. There are also those who willfully attempt to overwrite the ends of arrays in an attempt to breach the security of a system. Without mechanisms to enforce these memory rules, they become simply guidelines and provide no sort of security at all.

This necessary enforcement happens at three different times in the development and deployment of a Java program: at compile time, at link time (that is, when a class is loaded into the virtual machine), and at runtime. Not all rules can be checked at each of these points, but certain checks are necessary at each point in order to ensure the memory security that we're after. As we'll see, enforcement of these rules (which is really the construction of this part of the Java sandbox) varies depending on the origin of the class in question.

3.2.1 Compiler Enforcement

The Java compiler is the first thing that is tasked with the job of enforcing Java's language rules. In particular, the compiler is responsible for enforcing all of the rules we outlined above except for the last two: the compiler cannot enforce array bound checking nor can it enforce all cases of illegal object casts.

The compiler does enforce certain cases of illegal object casts −− namely, casts between objects that are known to be unrelated, such as the following code:

```
Vector v = new Vector(  );
String s = (String) v;
```

But the validity of a cast between an object of type X to type Y where Y is a subclass of X cannot be known at compile time, so the compiler must let such a construct pass.

3.2.2 The Bytecode Verifier

Okay, the compiler has produced a Java program for us, and we're about to run the Java bytecode of that program. But if the program came from an unknown source, how do we know that the bytecodes we've received are actually legal?

Bytecode Verification of Other Languages

Throughout this section, we're discussing the bytecode verifier as if it were tied to the Java language. This is somewhat imprecise: the bytecode verifier is actually independent of the original source language of the program. If we had a C++ compiler that generated Java bytecodes from C++ source, the bytecode verifier would still be able to verify (or not) the bytecodes.

However, the verification of the bytecodes would still depend upon the semantics of the Java language, and not the semantics of C++; just because the bytecodes in question originated from C++ code is no reason that they should suddenly be allowed to cast an arbitrary memory location into an object.

For this reason, I prefer to think of the bytecodes in terms of the Java language itself. There are tools to produce Java bytecodes from other languages (like Scheme), but in general, producing Java bytecodes from another language severely limits the constructs that can be written in that other language.

This brings us to the need for the bytecode verifier −− the second link in the chain of responsibility of enforcing the rules of the Java language. Normally when the need for the bytecode verifier is discussed, it's in terms of an evil compiler −− that is, a compiler that someone has written in such a way that the code produced by the compiler is not legal Java code. The theory is that code from such a compiler could be constructed in order to create and exploit a security hole by ignoring a rule in the Java language. Such an attack might seem to be difficult to achieve in that it would require some detailed knowledge of the Java compiler.

It turns out that the evil compiler issue is a red herring −− it doesn't really matter whether such an attack is likely or not because it's trivial to create nonconforming Java code with any standard Java compiler. Assume that we have these two classes:

```
package javasec.samples.ch03;

public class CreditCard {
    public String acctNo = "0001 0002 0003 0004";
}

package javasec.samples.ch03;

public class Test {
    public static void main(String args[]) {
        CreditCard cc = new CreditCard(  );
        System.out.println("Your account number is " + cc.acctNo);
    }
}
```

If we run this code, we'll create a `CreditCard` object and print out its account number. Now say that we realize the account number should really have been private, so we go back and change the definition of `acctNo` to be private and recompile only the `CreditCard` class. We then have two class files and the `Test` class file contains Java code that illegally accesses the private instance variable `acctNo` of the `CreditCard` class.

The above example shows an innocent mistake, but a malicious programmer could use just this technique to produce illegal Java bytecodes. In order to modify the contents of a string, for example, all we need to do is:

1. Copy the `java.lang.String` source file into our `classpath`.
2. In the copy of the file, modify the definition of `value` −− the private array that holds the actual characters of the string −− to be public.
3. Compile this modified class, and replace the class files in the SDK. This entails unpacking *rt.jar*, copying in the new classes, and repacking *rt.jar*.
4. Compile some new code against this modified version of the `String` class. The new code could include something like this:

```
package javasec.samples.ch03;

public class CorruptString {
    public static void modifyString(String src, String dst) {
        for (int i = 0; i < src.value.length; i++) {
            if (i == dst.value.length)
                return;
            src.value[i] = dst.value[i];
        }
    }

    public static void main(String[] args) {
        System.out.println("Original string is " + args[0]);
        modifyString(args[0], "SDO was here");
```

```
            System.out.println("Modified string is " + args[0]);
        }
    }
```

Now any time you want to modify a string in place, simply call this `modifyString( )` method with the string you want to corrupt (`src`) and the new string you want it to have (`dst`).

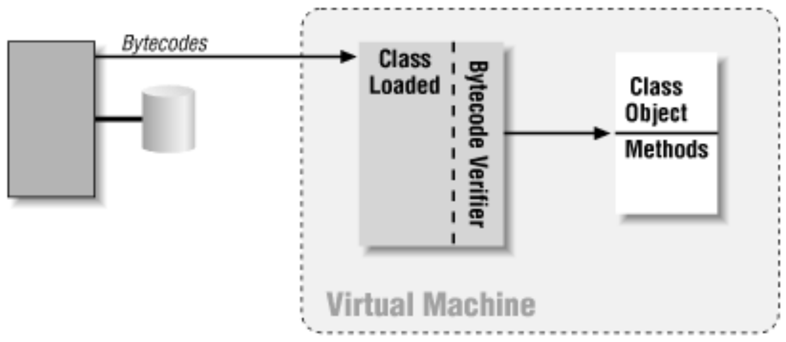
5. Remove the modified version of the `String` class.

Now the `CorruptString` class can be referenced by a Java program, which can use it to attempt to corrupt any string that it has a reference to. Even though the program will run with the original version of the `String` class, the `CorruptString` class will be able to access the private value array within the `String` class −− unless the bytecode verifier rejects the `CorruptString` class.

3.2.2.1 Inside the bytecode verifier

The bytecode verifier is an internal part of the Java virtual machine and has no interface: programmers cannot access it and users cannot interact with it. The verifier automatically examines most bytecodes as they are built into class objects by the class loader of the virtual machine (see Figure 3−1). We'll give just a brief overview of how the bytecode verifier actually works.

Figure 3−1. The bytecode verifier



The verifier is often referred to as a mini−theorem prover (a term first used in several documents from Sun). This sounds somewhat more impressive than it is; it's not a generic, all−purpose theorem prover by any means. Instead, it's a piece of code that can prove one (and only one) thing −− that a given series of (Java) bytecodes represents a legal set of (Java) instructions.

Specifically, the bytecode verifier can prove the following:

- The class file has the correct format. The full definition of the class file format may be found in the Java virtual machine specification; the bytecode verifier is responsible for making sure that the class file has the right length, the correct magic numbers in the correct places, and so on.
- Final classes are not subclassed, and final methods are not overridden.
- Every class (except for `java.lang.Object`) has a single superclass.
- There is no illegal data conversion of primitive data types (e.g., `int` to `Object`).
- No illegal data conversion of objects occurs. Because the casting of a superclass to its subclass may be a valid operation (depending on the actual type of the object being cast), the verifier cannot ensure that such casting is not attempted −− it can only ensure that before each such attempt is made, the legality of the cast is tested.
- There are no operand stack overflows or underflows.

  In Java, there are two stacks for each thread. One stack holds a series of method frames, where each method frame holds the local variables and other storage for a particular method invocation. This stack is known as the data stack and is what we normally think of as the stack within a traditional program. The bytecode verifier cannot prevent overflow of this stack −− an infinitely recursive method call will cause this stack to overflow. However, each method invocation requires a second stack (which itself is allocated on the data stack) that is referred to as the operand stack; the operand stack holds the values that the Java bytecodes operate on. This secondary stack is the stack that the bytecode verifier can ensure will not overflow or underflow.

Hence, when the bytecode verifier has completed its task, we know that the code in question follows many of the constraints of the Java language −− including most of the rules that the compiler was also responsible for ensuring. The remaining rules are verified during the actual running of the program.

3.2.2.2 Delayed bytecode verification

When we began this section, we said that the bytecode verifier is responsible for *examining* all the bytecodes of the class −− we explicitly did not say that the verifier is responsible for *verifying* all the bytecodes. This is because the bytecode verifier may delay some of the checks it is responsible for, as long as those checks are performed before the code is actually executed. In typical verifier implementations, the bytecode verifier does not immediately test to see if all field and method accesses are legal according to the access modifiers associated with that field or method.

This is driven by a desire to be efficient −− our `Test` class may reference the `acctNo` field of our `CreditCard` class, but it may do so only if a particular branch in the code is taken. In the following code, there's no need to verify that the access to `acctNo` is legal unless an `IllegalArgumentException` has been generated:

```
CreditCard cc = getCreditCard(  );
try {
        Wallet.makePurchase(cc);
} catch (IllegalArgumentException iae) {
        System.out.println("Can't process for account " + cc.acctNo);
}
```

Hence, the bytecode verifier delays all tests for field and method access until the code is actually executed. The process by which this happens is implementation independent; one technique that is often used is to ensure during verification that all accesses test the validity of the field access. If the access is valid, the standard bytecodes are then replaced during execution with a special bytecode indicating that the test has been performed and access to the field in question no longer needs to be tested. On the other hand, if the validity test fails, the virtual machine throws an `IllegalAccessException`.

This gives us the best of both worlds −− verification of the access is performed during the actual running of the program (after traditional bytecode verification has occurred), but the verification is still only performed once (unlike the runtime verification we'll examine later).

3.2.3 Runtime Enforcement

Like the compiler, the bytecode verifier cannot completely guarantee that the bytecodes follow all of the rules we outlined earlier in this chapter: it can only ensure that the first four of them are followed. The virtual machine must still take responsibility for ultimately determining that the Java bytecodes provide the security we expect them to.

The remaining security protections of the Java language must be enforced at runtime by the virtual machine.

Array bounds checking

In theory, the bytecode verifier can detect certain cases of array bounds checking, but in general, this check must take place at runtime. Consider the following code:

```
void initArray(int a[], int nItems) {
        for (int i = 0; i < nItems; i++) {
                a[i] = 0;
        }
}
```

Since `nItems` and `a` are parameters, the bytecode verifier has no way of determining whether this code is legal. Hence, array bounds checking is always done at runtime. Failure to meet this rule results in an `ArrayIndexOutOfBoundsException`.

Object casting

The verifier can and will detect the legality of certain types of casts, specifically whenever unrelated classes are cast to each other. The virtual machine must monitor when a superclass is cast into a subclass and test that cast's validity; failure to execute a legal cast results in a `ClassCastException`. This holds for casts involving interfaces as well since objects that are defined as an interface type (rather than a class type) are considered by the verifier to be of type `Object`.

3.3 Comparisons with Previous Releases

At the level of the Java language, little has changed between Java 1.1 and any release of the Java 2 platform. One thing which has changed is bytecode verification: in 1.1, the virtual machine did not perform bytecode verification of classes on the classpath. This is really a reflection of class loading policies: starting with the Java 2 platform, classes on the classpath are loaded by a traditional class loader, which subjects them to verification.

3.3.1 Controlling Bytecode Verification

Bytecode verification seems like a great thing: not only can it help to prevent malicious attacks from violating rules of the Java language, it can also help detect simple programmer errors -- such as when we changed the access modifier of `acctNo` in our `CreditCard` class but forgot to recompile our `Test` class.

Nonetheless, in 1.1 only classes loaded by a browser are subject to bytecode verification. In typical usage, this is a workable policy. Browsers always ensure that the code imported to run an applet is verified, and Java applications are typically not verified. Of course, this may or may not be the perfect solution:

- If a remote site can talk an end user into installing a local class in the browser's `classpath`, the local class will not be verified and may violate the rules we've discussed here. In Java 2, this is much harder since the class must be added to the jar file containing the core API classes.
- You may implicitly rely upon the verifier to help you keep files in sync so that when one is changed, other files are verified against it.

As a user, you (theoretically) have limited control over the verifier -- though such control depends on the browser you are using. If you are running a Java application, you can run `java` with the `-verify` option, which will verify all classes. Similarly, if you are using a browser written in Java -- including the `appletviewer` -- you can arrange for the `java` command to run with the `-noverify` option, which turns verification off for all classes. Occasionally, a browser not written in Java will allow the user to disable bytecode verification as well -- e.g., Internet Explorer 3.0 for the Mac had this capability, although it was present only because the bytecode verifier could not run in certain limited memory configurations.

However, although these options to the virtual machine are well−documented, they are not implemented on all platforms (and they have not survived in Java 2). One way to ensure that application code is run through the bytecode verifier in 1.1 is to use the secure launcher from Appendix D.

3.4 Summary

Because the notion of security in Java is pervasive, its implementation is equally pervasive. In this chapter, we've explored the security mechanisms that are built into the Java language itself. Essentially, at this level the security mechanisms are concerned with establishing a set of rules for the Java language that creates an environment where an object's view of memory is well−known and well−defined so that a developer can ensure that items in memory cannot be accidentally or intentionally read, corrupted, or otherwise misused. We also took a brief look at Java's bytecode verifier, including why it is necessary and why you should turn it on, even for Java applications.

It's important to keep in mind that the purpose of these security constraints is to protect the user's machine from a malicious piece of code and not to protect a piece of code from a malicious user. Java does not (and could not) prevent a user from acting on memory from outside the browser, with possibly harmful results.

Chapter 4. The Security Manager

In the next three chapters, we're going to discuss how the sandbox of a Java application is implemented. The implementation of the sandbox depends on three things:

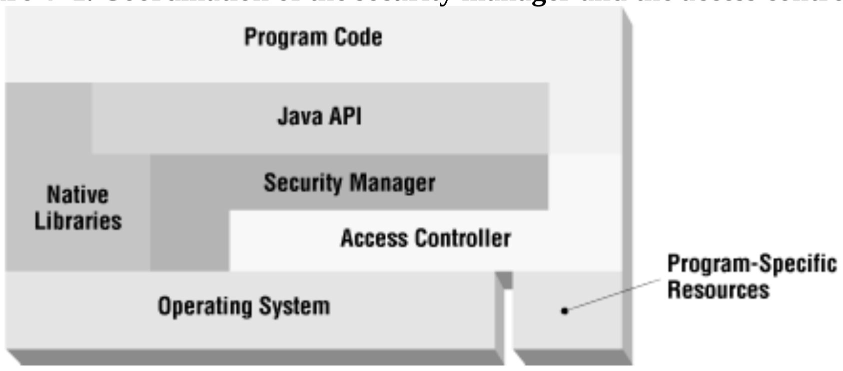
- The security manager, which provides the mechanism that the Java API uses to see if security−related operations are allowed.
- The access controller, which provides the basis of the default implementation of the security manager.
- The class loader, which encapsulates information about security policies and classes.

We'll start by examining the security manager. From the perspective of the Java API, there is a security manager that actually is in control of the security policy of an application. The purpose of the security manager is to determine whether particular operations should be permitted or denied. In truth, the purpose of the access controller is really the same: it decides whether access to a critical system resource should be permitted or denied. Hence, the access controller can do everything the security manager can do.

The reason there is both an access controller and a security manager is mainly historical: the access controller is only available in Java 2 and subsequent releases. Before the access controller existed, the security manager relied on its internal logic to determine the security policy that should be in effect, and changing the security policy required changing the security manager itself. Starting with Java 2, the security manager defers these decisions to the access controller. Since the security policy enforced by the access controller can be specified by using policy files, this allows a much more flexible mechanism for determining policies. The access controller also gives us a much simpler method of granting fine−grained, specific permissions to specific classes. That process was theoretically possibly with the security manager alone, but it was simply too hard to implement.

But the large body of pre−Java 2 programs dictates that the primary interface to system security −− that is, the security manager −− cannot change; otherwise, existing code that implements or depends on the security manager would become obsolete. Hence, the introduction of the access controller did not replace the security manager −− it supplemented the security manager. This relationship is illustrated in Figure 4−1. Typically, an operation proceeds through the program code into the Java API, through the security manager to the access controller, and finally into the operating system. In certain cases, however, the security manager may bypass the access controller. And native libraries are still outside the domain of either the security manager or the access controller (although the ability to load those libraries may be restricted).

Figure 4−1. Coordination of the security manager and the access controller



4.1 Overview of the Security Manager

When most people think of Java security, they think of the protections afforded to a Java program −− and, more particularly, only by default to a Java applet −− by Java's security manager. There are other important facets of Java's security story, but the role played by the security manager is of paramount importance in defining the security policy under which a particular program will operate.

On one level, the Java security manager is simple to understand, and it's often summarized by saying that it prevents Java applets from accessing your local disk or local network. The real story is more complicated than that, however, with the result that Java's security manager is often misunderstood.

On a simple level, the security manager is responsible for determining most of the parameters of the Java sandbox −− that is, it is ultimately up to the security manager to determine whether many particular operations should be permitted or rejected. If a Java program attempts to open a file, the security manager decides whether or not that operation should be permitted. If a Java program wants to connect to a particular machine on the network, it must first ask permission of the security manager. If a Java program wants to alter the state of certain threads, the security manager will intervene if such an operation is considered dangerous.

The security manager is used only if it is explicitly installed. When you run a Java application, specifying the `−Djava.security.manager` option installs a security manager. The security manager is installed programatically by the appletviewer and the Java Plug−in.

Hence, this point cannot be overemphasized: Java applications (at least by default) have no security manager while Java applets (again, by default) have a very strict security manager. This leads to a common misconception that exists in the arena of Java security: it's common to think that because Java is said to be secure, it is always secure and that running Java applications that have been installed locally is just as secure as running Java applets inside a Java−enabled browser. Nothing is further from the truth.

To illustrate this point, consider the following malicious code:

```
package javasec.samples.ch04;

import java.applet.*;

public class MaliciousApplet extends Applet {
    public void init(  ) {
        try {
            Runtime.getRuntime(  ).exec("rmdir foo");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    public static void main(String args[]) {
        MaliciousApplet a = new MaliciousApplet(  );
        a.init(  );
    }
}
```

If you compile this code, place it on your web server, and load it as an applet, you will get an error reflecting a security violation. However, if you compile this code, place it in a directory, and then run it as an application (without using the `−Djava.security.manager` option), you'll end up deleting the directory named *foo* in your current directory.[1] As a user, then, it's crucial that you understand which security manager is in place when you run a Java program so that you understand just what types of operations you are protected against.

[1] Fortunately, for testing, this code doesn't execute `/bin/rm −r` or `del *.*`.

4.1.1 Security Managers and the Java API

The security manager is a partnership between the Java API and the implementor of a specific Java application or of a specific Java−enabled browser. There is a class in the Java API called `SecurityManager` (`java.lang.SecurityManager`) which is the linchpin of this partnership −− it provides the interface that the rest of the Java API uses to check whether particular operations are to be permitted. The essential algorithm the Java API uses to perform a potentially dangerous operation is always the same:

1. The programmer makes a request of the Java API to perform an operation.
2. The Java API asks the security manager if such an operation is allowable.
3. If the security manager does not want to permit the operation, it throws an exception back to the Java API, which in turn throws it back to the user.
4. Otherwise, the Java API completes the operation and returns normally.

Let's trace this idea with an example:

```
package javasec.samples.ch04;

import java.io.*;

public class Cat {
    public static void main(String args[]) {
        try {
            String s;
            FileReader fr = new FileReader(args[0]);
            BufferedReader br = new BufferedReader(fr);
            while ((s = br.readLine(  )) != null)
                System.out.println(s);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

The `FileReader` object will in turn create a `FileInputStream` object, and constructing the input stream is the first step of the algorithm. When the input stream is constructed, the Java API performs code similar to this:

```
public FileInputStream(String name) throws FileNotFoundException {
    SecurityManager security = System.getSecurityManager(  );
    if (security != null) {
        security.checkRead(name);
    }
    try {
        open(name);         // open(  ) is a private method of this class
    } catch (IOException e) {
        throw new FileNotFoundException(name);
    }
}
```

This is step two of our algorithm and is the essence of the idea behind the security manager: when the Java API wants to perform an operation, it first checks with the security manager and then calls a private method (the `open(  )` method in this case) that actually performs the operation.

Meanwhile, the security manager code is responsible for deciding whether or not the file in question should be allowed to be read and, if not, for throwing a security exception:

```
public class SecurityManagerImpl extends SecurityManager {
    public void checkRead(String s) {
        if (theFileIsNotAllowedToBeRead)
            throw new SecurityException("Not allowed to read " + s);
    }
}
```

The `SecurityException` class is a subclass of the `RuntimeException` class. Remember that runtime exceptions are somewhat different than other exceptions in Java in that they do not have to be caught in the code −− which is why the `checkRead( )` method does not have to declare that it throws that exception, and why the `FileInputStream` constructor does not have to catch it. So if the security exception is thrown by the `checkRead( )` method, the `FileInputStream` constructor will return before it calls the `open( )` method −− which is simply to say that the input file will never be opened because the security manager prevented that code from being executed.

Typically, the security exception propagates up through all the methods in the thread that made the call; eventually, the top−most method receives the exception, which causes that thread to exit. When the thread exits in this way, it prints out the exception and the stack trace of methods that led it to receive the exception. This leads to the messages that you've probably seen in your Java console:

```
sun.applet.AppletSecurityException: checkread
        at sun.applet.AppletSecurity.checkRead(AppletSecurity.java:427)
        at java.io.FileOutputStream.<init>(FileOutputStream.java)
        at Cat.init(Cat.java:7)
        at sun.applet.AppletPanel.run(AppletPanel.java:273)
        at java.lang.Thread.run(Thread.java)
```

If the security exception is not thrown −− that is, if the security manager decides that the particular operation should be allowed −− then the method in the security manager simply returns and everything proceeds as expected.

Several methods in the `SecurityManager` class are similar to the `checkRead( )` method. It is up to the Java API to call those methods at the appropriate time. You may want to call those methods from your own Java code (using the technique shown in the previous example), but that's never required. Since the Java API provides the interface to the virtual operating system for the Java program, it's possible to isolate all the necessary security checks within the Java API itself.

You Don't Know About All Security Violations

Since a violation of the rules of the security manager manifests itself as a security exception, it's possible to hide the attempted operation from the user running the program by catching that exception.

To portray this feature in a positive light, it allows the developer to provide a more intelligent program that might be delivered to an end user in different ways. If the program is delivered as an application, the author may want to save some state from the program in a file on the user's disk; if the program is delivered as an applet, the author will need to save that state by sending it to the web server. The program might have code that looks like this:

```
OutputStream os;
try {
    os = new FileOutputStream("statefile");
} catch (SecurityException e) {
    os = new Socket(webhost, webport).getOutputStream(  );
}
```

Now the Java program has an appropriate output stream to save its data.

On the other hand, this technique can be used by the author of an applet to probe your browser's security manager without your knowledge −− and because the applet is catching the security exceptions, you'll never see them. This is one reason why it's important to understand the ramifications of adjusting your browser's security policy.

One exception to this guideline occurs when you extend the virtual operating system of the Java API, and it is important to ensure that your extensions are well−integrated into Java's security scheme. Certain parts of the Java API −− the `Toolkit` class, the `Provider` class, the `Socket` class, and others −− are written in such a way that they allow you to provide your own implementation of those classes. If you're providing your own implementation of any one of these classes, you have to make sure that it calls the security manager at appropriate times.

It's important to note that there is (by design) no attempt in the Java API to maintain any sort of state. Whenever the Java API needs to perform an operation, it checks with the security manager to see if the operation is to be allowed −− even if that same operation has been permitted by the security manager before. This is because the context of the operation is often significant −− the security manager might allow a `FileOutputStream` object to be opened in some cases (e.g., by certain classes) while it might deny it in other cases. The Java API cannot keep track of this contextual information, so it asks the security manager for permission to perform every operation.

4.2 Operating on the Security Manager

There are two methods in the `System` class that are used to work with the security manager itself:

public static SecurityManager getSecurityManager( )

Return a reference to the currently installed security manager object (or `null` if no security manager is in place). Once obtained, this object can be used to test against various security policies.

public static void setSecurityManager(SecurityManager sm)

Set the system's security manager to the given object. Code that wants to install a security manager must have the runtime permissions `createSecurityManager` in order to instantiate the security manager object and `setSecurityManager` in order to install it.

These methods operate with the understanding that there is a single security manager in the virtual machine; the only operations that are possible on the security manager are setting it (that is, creating an instance of the security manager class and telling the virtual machine that the newly created object should be the security manager) and getting it (that is, asking the virtual machine to return the object that is the security manager so that a method might be invoked upon it).

We've already seen how you might use the `getSecurityManager( )` method to retrieve the security manager and invoke an operation on it. Setting the security manager is a predictably simple operation:

```
public static void main(String args[]) {
    System.setSecurityManager(new SecurityManagerImpl(  ));
        ... do the work of the application ...
    }
}
```

The `SecurityManager` class provides a complete implementation that uses the access controller to implement the permission−based sandbox we discussed in Chapter 2. When you specify the `−Djava.security.manager` option to a Java application, the virtual machine executes the `setSecurityManager( )` method on your behalf, before it calls the `main( )` method of your application.

However, it's possible to extend the `SecurityManager` class to provide a different implementation of the sandbox. The Java Plug−in and `appletviewer` use such a modified implementation and install it before they load any applets. And while the changes that the Java Plug−in and `appletviewer` make are very minor, other environments can have completely different implementations.

4.3 Methods of the Security Manager

Now that we have an understanding of how the security manager works, we'll look into what protection the security manager actually provides. We'll discuss the public methods of the security manager that perform security checks and when those methods are called, along with the rationale behind each of the methods. Since these methods are all public, they can be called anywhere, including in your own code, although as we've mentioned, that's a rare thing. The real point of this section is so that you can know which methods of the core Java API are affected by the security manager and to give you some background on why these choices were made.

When we discuss the methods below, we speak of them in terms of trusted and untrusted classes. A trusted class is a class of the core Java API or a class that has been granted explicit permission to perform the operation in question.

You'll note that the methods of the security manager correspond fairly well to the set of default permissions that we listed in Chapter 2. In fact, the basic implementation of each method of the security manager is to test to ensure that each active protection domain has permission to perform the desired operation.

4.3.1 Methods Relating to File Access

The most well−known methods of the security manager class handle access to files on the local network. This includes any files that are on the local disk as well as files that might be physically located on another machine but appear (through the use of NFS, NetWare, Samba, or a similar network−based filesystem) to be part of the local filesystem.

These are the methods the security manager uses to track file access:

public void checkRead(FileDescriptor fd)
public void checkRead(String file)
public void checkRead(String file, Object context)

Check whether the program is allowed to read the given file. The first of these methods succeeds if the current protection domain has been granted the runtime permission with the name `readFileDescriptor`. The other methods succeed if the current protection domain has a file permission with a name that matches the given file and an action of read.

public void checkWrite(FileDescriptor fd)
public void checkWrite(String file)

Check whether the program is allowed to write the given file. The first of these methods succeeds if the current protection domain has been granted the runtime permission with the name `writeFileDescriptor`. The second succeeds if the current protection domain has a file permission with a name that matches the given file and an action of write.

public void checkDelete(String file)

Check whether the program is allowed to delete the given file. This succeeds if the current protection domain has a file permission with a name that matches the given file and an action of delete.

Interestingly, although as developers we tend to think of other file operations −− such as creating a file or seeing when the file was last modified −− as being distinct operations, as far as security is concerned, the Java API considers all operations to be either reading, writing, or deleting.

Table 4−1 lists the Java API interaction with the `checkRead( )`, `checkWrite( )`, and `checkDelete( )` methods, listing when and why each check is invoked. In all the tables in this chapter, the syntax may imply that the calling methods are all static, but it of course is not the case: the entry `File.canRead( )` means the `canRead( )` method invoked on an instance of the `File` class.

This table lists only those classes that directly call the security manager method in question. There may be many routes through the Java API that lead to one of these checks; for example, when a `FileReader` object is constructed, it will construct a `FileInputStream` object, which will result in a call to `checkRead( )`.

Table 4−1. Check Methods

Method	Called By	Rationale
`checkRead( )`	`File.canRead( )`	Test if the current thread can read the file.
	`FileInputStream( )` `RandomAccessFile( )`	Constructing a file object requires that you read the file.
	`File.isDirectory( )` `File.isFile( )`	Determining whether a file object is an actual file or a directory requires that you read the file.
	`File.lastModified( )`	Determining the modification date requires that you read the file's attributes.
	`File.length( )`	Determining the length requires that you read the file's attributes.
	`File.list( )`	Determining the files in a directory requires that you read the directory.
`checkWrite( )`	`File.canWrite( )`	Test if the current thread can write the file.
	`FileOutputStream( )` `RandomAccessFile( )`	To construct a file object, you must be able to write the file.
	`File.mkdir( )`	To create a directory, you must be able to write to the filesystem.
	`File.renameTo( )`	To rename a file, you must be able to write to the directory containing the file.
	`File.createTempFile( )`	To create a temporary file, you must be able to write the file.
`checkDelete( )`	`File.delete( )`	Test if the current thread can delete a file.
	`File.deleteOnExit( )`	Test if the current thread can delete the file when the virtual machine exits.

If you carefully considered the list of methods in the table above, you were probably surprised not to see an obvious method to check: the actual `read( )` or `write( )` methods of any of the `File` classes. The assumption here is that a trusted class is responsible for determining the security policy associated with any particular `File` object; if the trusted class decides that it is okay for an untrusted class to perform I/O on a particular `File*Stream` object, then it is free to deliver that object to the untrusted class, and the untrusted class is free to read or write to that object. This implementation also allows for much greater efficiency: if the program had to check with the security manager every time it called the `read( )` or `write( )` methods, I/O performance would drastically suffer.

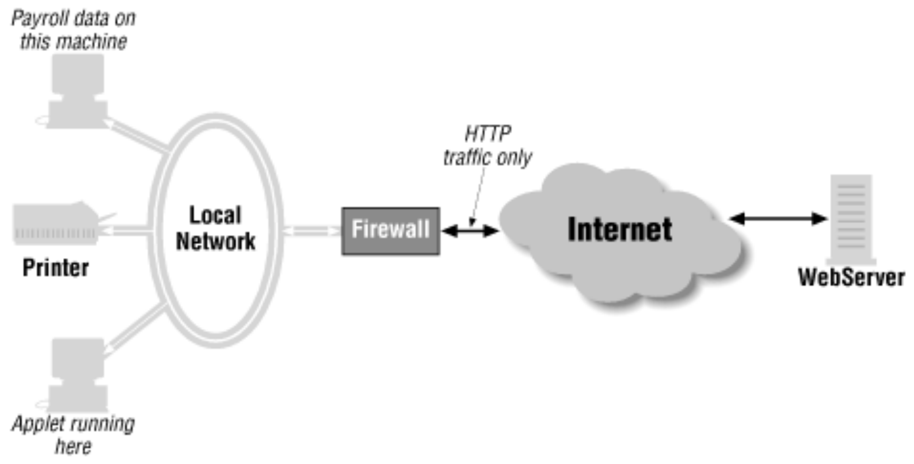
4.3.2 Methods Relating to Network Access

Network access in Java is always accomplished by opening a network socket, whether directly through the `Socket` class or indirectly through another class like the `URL` class. An untrusted class can only (by default) open a socket to the machine from which it was actually downloaded; typically, this is the location given by the `CODEBASE` tag in the HTML for the browser page containing the applet or −− in the absence of such a tag −− the web server for the page. In either case, the machine in question is a web server, so we'll use that terminology in this discussion. Untrusted classes loaded from the classpath cannot, by default, open any sockets.

This restriction on untrusted classes is designed to prevent two types of attack. The first attack concerns a rogue applet using your machine for malicious purposes by connecting to a third machine over the network. The canonical description of this attack is an applet that connects to the mail server on someone else's machine and sends people on that machine offensive email from your address. There are more severe attacks possible with this technique, however −− such an applet could use a connection from your machine to break into a third computer, and auditors on that third computer will think the break−in attempts are coming from you, which can cause you all sorts of legal problems.

The second sort of attack concerns network information on your local network that you might not want to be broadcast to the world at large. Typically, computers at corporations or campuses sit behind a firewall so that users on the Internet cannot access those computers (see Figure 4−2). The firewall allows only certain types of traffic through (e.g., HTTP traffic) so that users on the local network can access the Internet, but users on the Internet cannot glean any information about the local network.

Figure 4−2. A typical firewall configuration



Now consider what happens if an applet downloaded onto a machine on the local network can connect to other machines on the local network. This allows the applet to gather all sorts of information about the local network topology and network services and to send that information (via HTTP, so that it will pass through the firewall) back out onto the Internet. Such an opportunity for corporate spying would be very tempting to would−be hackers. Worse, if the applet had access to arbitrary network services, it could break into the local HR database and steal employee data, or it could break into a network file server and steal corporate documents. Hence, applets (and untrusted classes in general) are prevented from arbitrary network access.

Network sockets can be logically divided into two classes: client sockets and server sockets. A client socket is responsible for initiating a conversation with an existing server socket; server sockets sit idle waiting for these requests to come from client sockets. Untrusted classes are by default restricted from creating server sockets.[2] Normally, this is not a problem: since an applet can only talk to its web server, it could only answer requests from that machine −− and the applet can already open a connection to that machine at will. There's no algorithmic or logistic reason why an operation between the applet and the web server cannot always start with the applet as the client.

[2] Technically, untrusted classes by default can create a server socket, since the default policy file allows all classes to perform the listen action. Once created, however, the server socket must accept a connection on the socket, and that action is by default denied.

The security manager uses the following methods to check network access:

public void checkConnect(String host, int port)
public void checkConnect(String host, int port, Object context)

Check if the program can open a client socket to the given port on the given host. This succeeds if the current protection domain has a socket permission with a name that matches the host and port and an action of connect.

public void checkListen(int port)

Check if the program can create a server socket that is listening on the given port. The protection domain must have a socket permission where the host is localhost, the port range includes the given port, and the action is listen.

public void checkAccept(String host, int port)

Check if the program can accept (on an existing server socket) a client connection that originated from the given host and port. The protection domain must have a socket permission where the host and port match the parameters to this method and an action of accept.

public void checkMulticast(InetAddress addr)
public void checkMulticast(InetAddress addr, byte ttl)

Check if the program can create a multicast socket at the given multicast address (optionally with the given time−to−live value). This succeeds if the current protection domain has a socket permission with a host that matches the given address, a port range of all ports, and an action list of connect and accept.

public void checkSetFactory( )

Check if the program can change the default socket implementation. When the `Socket` class is used to create a socket, it gets a new socket from the socket factory, which typically supplies a standard TCP−based socket. However, a program could install a socket factory so that the sockets it uses all have different semantics, such as sockets that send tracing information when they write data. The current protection domain must carry a runtime permission with the name `setFactory`.

The instances where these methods are used and the rationale for such uses are shown in Table 4−2.

Table 4−2. Security Manager Methods to Protect Network Access

Method	Called By	Rationale

`checkConnect(  )`	`DatagramSocket.send(  )`
	`DatagramSocket.receive(  )`
	`    (Deprecated)`
	`MulticastSocket.send(  )`
	`Socket(  )`
		Test if the untrusted class can create a client−side connection.

`checkConnect(  )`	`DatagramSocket.getLocalAddress(  )`
	`InetAddress.getHostName(  )`
	`InetAddress.getLocalHost(  )`
	`InetAddress.getAllByName(  )`
		Test if the untrusted class can see any hosts on the local network.

`checkListen(  )`	`DatagramSocket(  )`
	`MulticastSocket(  )`
	`ServerSocket(  )`
		Test if the untrusted class can create a server−side socket.

`checkMulticast(  )`	`DatagramSocket.send(  )`
	`DatagramSocket.receive(  )`
	`MulticastSocket.send(  )`
	`MulticastSocket.receive(  )`
	`MulticastSocket.joinGroup(  )`
	`MulticastSocket.leaveGroup(  )`
		Test if the untrusted class can operate on a multicast socket.

`checkAccept(  )`	`ServerSocket.accept(  )`
	`DatagramSocket.receive(  )`	

Test if the untrusted class can accept a server connection.

```
checkSetFactory(  )
```

```
ServerSocket.setSocketFactory(  )
Socket.setSocketFactory(  )
URL.setURLStreamHandlerFactory(  )
URLConnection.setContentHandlerFactory(  )
RMI.setSocketFactory(  )
```

Test if the untrusted class can alter the manner in which all sockets are created.

checkSetFactory( )

```
HttpURLConnection.setFollowRedirects(  )
```

Test if the untrusted class can change redirection behavior.

Some notes are in order. As in the case with file access, these methods sometimes check operations that are logically different from a programming view but are essentially the same thing from a system view. Hence, the `checkConnect( )` method not only checks the opening of a socket but also the retrieval of hostname or address information (on the theory that to know the name of a host, you need to be able to open a socket to that host). This last test may seem somewhat odd −− under what circumstances, you might wonder, should an untrusted class not be able to know the name or address of the machine on which it is running? Recall that we want to prevent the outside world from knowing our network topology; this includes the name and address of the user's machine as well.[3]

[3] On the other hand, there's a good chance the outside web server already has that information, since our browser sent along a hostname and other information when it retrieved the file to begin with. If our request passed through a firewall or proxy server, there's a chance that some of this information was prevented from passing to the outside web server, but that's not necessarily the case either.

The `checkSetFactory( )` method of the security manager class is responsible for arbitrating the use of several low−level aspects of Java's network classes. Most of the tests made by this method have to do with whether or not the untrusted class is allowed to create some variety of socket factory. Socket factories are classes that are responsible for creating sockets that implement a particular interface while having a nonstandard feature. For instance, you could use a socket factory to provide sockets that always perform encryption. Predictably, untrusted classes cannot change the socket factory in use.

This method is also used to determine whether the Java program will automatically follow redirect messages when opening a URL. When a Java program opens a URL, the server to which it is connected may send back a redirect response (an HTTP response code of 3xx). Often, browsers follow these redirects transparently to the user; in Java, the programmer has the ability to determine if the redirection should automatically be followed or not. An untrusted class is not able to change whether redirection is on or off. The `HttpURLConnection` class that uses this method is abstract, so the actual behavior of this class may be overridden in a particular implementation.

4.3.3 Methods Protecting the Java Virtual Machine

There are a number of methods in the `SecurityManager` class that protect the integrity of the Java virtual machine and the security manager. These methods fence in untrusted classes so that they cannot circumvent the protections of the security manager and the Java API itself. These methods are summarized in Table 4−3.

Table 4−3. Security Manager Methods Protecting the Virtual Machine

Method	Called By	Rationale
`checkCreateClassLoader( )`	`ClassLoader( )`	Class loaders are protected since they provide information to the security manager.
`checkExec( )`	`Runtime.exec( )`	Other processes might damage the user's machine.
`checkLink( )`	`Runtime.load( )` `Runtime.loadLibrary( )`	Don't let untrusted code import native code.
`checkExit( )`	`Runtime.exit( )`	Don't let untrusted code halt the virtual machine.
`checkExit( )`	`Runtime.runFinalizersOnExit( )`	Don't let untrusted code change whether finalizers are run.
`checkPermission( )`	Many	See if the current thread has been granted a particular permission.

public void checkCreateClassLoader( )

The distinction we keep mentioning between trusted and untrusted classes is often based on the location from which the class was loaded. As a result, the class loader takes on an important role since the security manager must ask the class loader where a particular class came from. The class loader is also responsible for marking certain classes as signed classes. Hence, an untrusted class is typically not allowed to create a class loader. This method is only called by the constructor of the `ClassLoader` class: if you can create a class loader (or if you obtain a reference to a previously created class loader), you can use it. To succeed, the current protection domain must have a runtime permission with the name `createClassLoader`.

public void checkExec(String cmd)

This method is used to prevent execution of arbitrary system commands by untrusted classes −− an untrusted class cannot, for example, execute a separate process that removes all the files on your disk.[4] To succeed, the current protection domain must have a file permission with a name that matches the given command and an action of execute.

[4] The separate process would not need to be written in Java, of course, so there would be no security manager around to enforce the prohibition about deleting files.

public void checkLink(String lib)

System commands aren't the only code that is out of reach of the security manager −− any native (C language) code that is executed by the virtual machine cannot be protected by the security manager (or, in fact, by any aspect of the Java sandbox). Native code is executed by linking a shared library into the virtual machine; this method prevents an untrusted class from linking in such libraries.

It may seem as if this check is very important. It is, but only to a point: the programmatic binding from Java to C is such that Java code cannot just call an arbitrary C function −− the C function must have a very specialized name that will not exist in an arbitrary library. So any C function that the untrusted class would like to call must reside in a library that you've downloaded and placed on your machine −− and if the program's author can convince you to do that, then you don't really have a secure system anyway and the author could find a different line of attack against you.

To succeed, the current protection domain must have a runtime permission with the name `loadLibrary.<lib>`.

public void checkExit(int status)

Next, there is the continuing processing of the virtual machine itself. This method prevents an untrusted class from shutting down the virtual machine. This method also prevents an untrusted class from changing whether or not all finalizers are run when the virtual machine does exit. This means that an untrusted class −− and in particular, an applet −− cannot guarantee that all the finalize methods of all the objects will be called before the system exits (which cannot be guaranteed in any case, since the browser can be terminated from the operating system without an opportunity to run the finalizers anyway).

When you install a security manager via the command−line argument, all code (trusted or not) is able to exit the virtual machine. In the `appletviewer` or the Java Plug−in, the current protection domain must have the runtime permission named `exitVM` in order for this call to succeed.

public void checkPermission(Permission p)
public void checkPermission(Permission p, Object context)

Check to see if the current thread has the given permission. This method is used when you write your own permission classes, as we'll examine in Chapter 5. It is also used directly by the Java API when it needs to test for runtime permissions. This method succeeds if the current protection domain has been granted the given permission.

4.3.4 Methods Protecting Program Threads

Java depends heavily on threads for its execution; in a simple Java program that uses images and audio, there may be a dozen or more threads that are created automatically for the user (depending on the particular implementation of the virtual machine). These are system−level threads responsible for garbage collection, the various input and output needs of the graphical interface, threads to fetch images, etc. An untrusted class cannot manipulate any of these threads because doing so would prevent the Java virtual machine from running properly, affecting other applets and possibly even the browser itself.

The security manager protects threads with these methods:

public void checkAccess(Thread g)
: Check if the program is allowed to change the state of the given thread. This call succeeds if the current protection domain has a runtime permission with the name `modifyThread`.

public void checkAccess(ThreadGroup g)
: Check if the program is allowed to change the state of the given thread group (and the threads that it holds). This call succeeds if the current protection domain has a runtime permission with the name `modifyThreadGroup`.

public ThreadGroup getThreadGroup( )
: Supply a default thread group for newly created threads to belong to.

Table 4−4 shows the methods of the Java API that are affected by the policy set in the `checkAccess( )` methods. By default, a thread is able to manipulate any other thread except for threads in the root thread group; it is able to manipulate any thread group except for the root thread group.

Table 4−4. Security Manager Methods Protecting Thread Access

Method	Called By	Rationale

`checkAccess(Thread g)`	`Thread.stop( )`
	`Thread.interrupt( )`
	`Thread.suspend( )`
	`Thread.resume( )`
	`Thread.setPriority( )`
	`Thread.setName( )`
	`Thread.setDaemon( )`
	`Thread.setClassLoader( )`
	`Thread( )`	

Untrusted classes may only manipulate threads that they have created.

```
checkAccess(ThreadGroup g)

ThreadGroup(  )
ThreadGroup.setDaemon(  )
ThreadGroup.setMaxPriority(  )
ThreadGroup.stop(  )
ThreadGroup.suspend(  )
ThreadGroup.resume(  )
ThreadGroup.destroy(  )
ThreadGroup.interrupt(  )
```

Untrusted classes can only affect thread groups that they have created.

```
getThreadGroup(  )

Thread(  )
```

Threads of untrusted classes must belong to specified groups.

```
checkPermission(Permission p)

Thread.stop(  )
```

Stopping a thread could corrupt state of the virtual machine.

Unlike the other public methods of the security manager, the `getThreadGroup( )` method is not responsible for deciding whether access to a particular resource should be granted or not, and it does not throw a security exception under any circumstances. The point of this method is to determine the default thread group that a particular thread should belong to. When a thread is constructed and does not ask to be placed into a particular thread group, the `getThreadGroup( )` method of the security manager is used to find a thread group to which the thread should be assigned. By default, this is the thread group of the calling thread.

The `getThreadGroup( )` method can be used to create a hierarchy of thread groups. One popular use of this method is to segregate applets loaded from different sites into their own thread group; in a server, it could be used to segregate the threads assigned to different clients into different thread groups. Doing so requires some cooperation with the class loader since it forms a natural boundary between different applets or different clients. However, a full hierarchy of thread groups does not mesh well with Java's default thread permission model, so we won't discuss that option in the main text. In Appendix D, we discuss the implementation of a different security manager that uses this notion of a thread group hierarchy.

The `Thread` class also calls the `checkPermission( )` method of the security manager whenever the `stop( )` method is called since stopping a thread is an inherently dangerous operation (which has led the `stop( )` method to become deprecated). For backward compatibility, this permission is normally granted even to untrusted classes, but an end user may change her environment so that the security manager throws an exception whenever the `stop( )` method is called.

4.3.5 Methods Protecting System Resources

The Java−enabled browser has access to certain system−level resources to which untrusted classes should not be granted access. The next set of methods (outlined in Table 4−5) in the `SecurityManager` class handles those system−level resources.

Table 4−5. Security Manager Protections of System Resources

Method	Called By	Rationale
`checkPrintJobAccess( )`	`Toolkit.getPrintJob( )`	Untrusted classes can't initiate print jobs.
`checkSystemClipboardAccess( )`	`Toolkit.getSystemClipboard( )`	Untrusted classes can't read the system clipboard.
`checkAwtEventQueueAccess( )`	`EventQueue.getEventQueue( )`	Untrusted classes can't manipulate window events.
`checkPropertiesAccess( )`	`System.getProperties( )` `System.setProperties( )`	Untrusted classes can't see or set system properties.
`checkPropertyAccess( )`	`System.getProperty( )`	Untrusted classes can't get a particular system property.
`checkPropertyAccess( )`	`Locale.setDefault( )`	Can't change the locale unless the `user.language` property can be read.

`checkPropertyAccess( )`

`Font.getFont( )`

Can't get a font unless its property can be read.

`checkTopLevelWindow( )`

`Window( )`

Windows created by untrusted classes should have an indentifying banner.

public void checkPrintJobAccess( )

Untrusted classes are not allowed access to the user's printer. This is another example of a nuisance protection; you wouldn't want a rogue applet sending reams of nonsense data to your printer. This method is never actually called by the standard Java API −− it's up to the platform−specific implementation of the AWT toolkit to call it.

Note this doesn't prevent the user from initiating a print action from the browser −− it only prevents an applet from initiating the print action. The utility of such a check is subtle: the user always has to confirm the print dialog box before anything is actually printed (at least with the popular implementations of the AWT toolkit). The only sort of scenario that this check prevents is this: the user could surf to *www.EvilSite.org* and then to *www.sun.com*; although the applets from *EvilSite* are no longer on the current page, they're still active, and one of them could pop up the print dialog. The user will associate the dialog with the *www.sun.com* page and presumably allow it to print −− and when the *EvilSite* applet then prints out offensive material, the user will blame the Sun page.

In order to succeed, the current protection domain must have an AWT permission with the name `queuePrintJob`.

public void checkSystemClipboardAccess( )

The Java virtual machine contains a system clipboard that can be used as a holder for copy−and−paste operations. Granting access to the clipboard to an untrusted class runs the risk that a class will come along, examine the clipboard, and find contents a previous program left there. Such contents might be sensitive data the new class should not be allowed to read; hence, untrusted classes are prevented from accessing the system clipboard. This restriction applies only to the system clipboard: an untrusted class can still create its own clipboard and perform its own copy−and−paste operations to that clipboard. Untrusted classes can also share non−system clipboards between them.

This method is also never actually called by the Java API; it's up to the platform−specific implementation of the AWT toolkit to call it. To succeed, the current protection domain must have an AWT permission of `accessClipboard`.

public void checkAwtEventQueueAccess( )

Similarly, the Java virtual machine contains a system event queue that holds all pending AWT events for the system. An untrusted class that had access to such a queue would be able to delete events from the queue or insert events into the queue. This protects against the same sort of scenario we saw for printing −− an applet on a previously visited page could insert events into the queue which would then be fed to an applet on the existing page.

Since this means that an untrusted class cannot get the system event queue, it is unable to call any of the methods of the `EventQueue` class −− specifically the `postEvent( )` and `peekEvent( )` methods. Note, however, that an applet may still post events to itself using the `dispatchEvent( )` method of the `Component` class.

To succeed, the current protection domain must carry an AWT permission of `accessEventQueue`. The default security manager implementation is overridden by the Java Plug−in and `appletviewer`, which allow applets access to these methods but filter out events from other applet codebases.

public void checkPropertiesAccess( )
public void checkPropertyAccess(String key)

The Java virtual machine has a set of global (system) properties that contains information about the user and the user's machine: login name, home directory, etc. Untrusted classes are generally denied access to some of this information in an attempt to limit the amount of spying that an applet can do. As usual, these methods only prevent access to the system properties; an untrusted class is free to set up its own properties and to share those properties with other classes if it desires.

To succeed, the current protection domain must carry a property permission. If a key is specified, then the name of the property permission must include the given key and have an action of read. Otherwise, the name of the property permission must be "*" and the action must be read and write.

public boolean checkTopLevelWindow(Object window)

Java classes, regardless of whether they are trusted or untrusted, are normally allowed to create top−level windows on the user's desktop. However, there is a concern that an untrusted class might bring up a window that looks exactly like another application on the user's desktop and thus confuse the user into doing something that ought not be done. For example, an applet could bring up a window that looks just like a telnet session and grab the user's password when the user responds to the password prompt. For that reason, top−level windows that are created by untrusted classes have some sort of identifying banner on them.

Note that unlike other methods in the security manager, this method has three outcomes: if it returns `true`, the window will be created normally; if it returns `false`, the window will be created with the identifying banner. Theoretically, this method could also throw a security exception (just like all the other methods of the security manager class) to indicate that the window should not be created at all; no popular implementations do that. In order for this method to return true, the current protection domain must carry an AWT permission with the name `showWindowWithoutWarningBanner`.

4.3.6 Methods Protecting Security Aspects

There are a number of methods in the security manager that protect Java's idea of security itself. These methods are summarized in Table 4−6.

Table 4−6. Security Manager Methods Protecting Java Security

Method

Called By

Rationale

```
checkMemberAccess(  )
```

```
Class.getFields(  )
Class.getMethods(  )
Class.getConstructors(  )
Class.getField(  )
Class.getMethod(  )
Class.getConstructor(  )
Class.getDeclaredClasses(  )
Class.getDeclaredFields(  )
Class.getDeclaredMethods(  )
Class.getDeclaredConstructors(  )
Class.getDeclaredField(  )
Class.getDeclaredMethod(  )
Class.getDeclardConstructor(  )
```

Untrusted classes can only inspect public information about other classes.

```
checkPackageAccess(  )
```

```
Not called
```

Check if the untrusted class can access classes in a particular package.

```
checkPackageDefinition(  )
```

```
Not called
```

Check if the untrusted class can load classes in a particular package.

```
checkSecurityAccess(  )
```

```
Identity.setPublicKey(  )
Identity.setInfo(  )
Identity.addCertificate(  )
Identity.removeCertificate(  )
IdentityScope.setSystemScope(  )
Provider.clear(  )
Provider.put(  )
Provider.remove(  )
Security.insertProviderAt(  )
Security.removeProvider(  )
Security.setProperty(  )
Signer.getPrivateKey(  )
Signer.setKeyPair(  )
```

Untrusted classes cannot manipulate security features.

public void checkMemberAccess(Class clazz, int which)

In Chapter 3, we examined the importance of the access modifiers to the integrity of Java's security model. Java's reflection API allows programs to inspect classes to determine the class's methods, variables, and constructors. The ability to access these entities can impact the memory integrity that Java provides.

The reflection API is powerful enough that, by inspection, a program can determine the private instance variables and methods of a class (although it cannot actually access those variables or call those methods). All classes are allowed to inspect any other class and find out about its public variables and methods. All classes loaded by the same class loader are allowed to inspect all of each other's variables and methods. Otherwise, the current protection domain must carry a runtime permission with a name of `accessDeclaredMembers`.

The default implementation of this method is very fragile. Unlike all other methods that we'll look at, it is a logical error to override this method and then call `super.checkMemberAccess( )`.

public void checkSecurityAccess(String action)

In the last half of this book, we'll be examining the details of the Java security package. This package implements a higher−order notion of security, including digital signatures, message digests, public and private keys, etc. The security package depends on this method in the security manager to arbitrate which classes can perform certain security−related operations. As an example, before a class is allowed to read a private key, this method is called with a string indicating that a private key is being read.

Predictably, only trusted classes are allowed to perform any of these security−related operations. Although the string argument gives the ability to distinguish what operation is being attempted, that argument is typically ignored in present implementations. As we discuss the features of the security package itself, we'll examine how the security package uses this method in more depth.

public void checkPackageAccess(String pkg)
public void checkPackageDefinition(String pkg)

These methods are used in conjunction with a class loader. When a class loader is asked to load a class with a particular package name, it will first ask the security manager if it is allowed to do so by calling the `checkPackageAccess( )` method. This allows the security manager to make sure that the untrusted class is not trying to use application−specific classes that it shouldn't know about.

Similarly, when a class loader actually creates a class in a particular package, it asks the security manager if it is allowed to do so by calling the `checkPackageDefinition( )` method. This allows the security manager to prevent an untrusted class from loading a class from the network and placing it into, for example, the `java.lang` package.

Notice the distinction between these two methods: in the case of the `checkPackageAccess()` method, the question is whether the class loader can reference the class at all −− e.g., whether we can call a class in the `sun` package. In the `checkPackageDefinition( )` method, the class bytes have been loaded and the security manager is being asked if they can belong to a particular package.

By default, the `checkPackageDefinition( )` method is never called and the `checkPackageAccess( )` method is called only for packages listed in the `package.access` property within the *java.security* file. If you write a class loader, you may call it as we indicate in Chapter 6. To succeed, the current protection domain must have a runtime permission with the name `defineClassInPackage.+<pkg>` or `accessClassInPackage.+<pkg>`.

That's all the methods of the security manager class that are used by the Java API to perform checks on certain operations. There are other public and protected methods of the `SecurityManager` class that we have not examined in this chapter; those methods are generally only used when you implement your own security manager without using the access controller, so we will defer their discussion to Appendix D. In the next chapter, we'll discuss how the security manager is usually implemented.

4.4 Comparison with Previous Releases

The security manager has existed in every release of Java. In Java 1.0 and 1.1, the security manager is the only thing that affects the security policy of the program. Because there is no way to install a default security manager via the command line prior to Java 2, most Java 1.0 and 1.1 applications do not have a security manager. In addition, the implementation of the security manager between 1.1−based browsers varies in important aspects between different browser vendors. Even though some browser vendors claim to support the Java 2 platform, they still implement their own security manager rather than using the permission and policy−based default security manager.

We'll discuss many of the major differences here. In addition, in Appendix D, we'll show how a security manager could be implemented in order to specify a policy for applications run in 1.1.

4.4.1 Trusted and Untrusted Classes

The default notion of what constitutes a trusted class has changed significantly between releases of Java:

- In Java 1.0, a class that is loaded from the classpath is considered trusted and a class that is loaded from a class loader is considered untrusted.
- In Java 1.1, the same rule applies but a class that is loaded from a jar file may carry with it a digital signature that allows it to be given extra privileges. These privileges are typically all−or−nothing: if you trust the entity that signed the jar file, then that code can do anything it wants. Some browser vendors have extended that behavior using proprietary APIs.
- In Java 2, only classes in the core API are considered trusted. Other classes must be given explicit permission to perform the operations we've discussed.

4.4.2 Differences in the Security Manager Class

In 1.1, the `setSecurityManager( )` method can only be called once, and once installed, the security manager cannot be removed. Attempting to call this method after a security manger has already been installed will result in a `SecurityException`.

4.4.2.1 File access

While 1.1−based browsers like Netscape Navigator 4 and earlier, Internet Explorer, and HotJava all have a default policy that prevents untrusted classes from all file access, some of them allow the user to configure a different policy. HotJava and the `appletviewer`, for example, allow the user to create a set of directories in which applets can read and write files, and some versions of Internet Explorer allow the user to grant file access to all untrusted classes.

4.4.2.2 Network access

There was a change in the default security policy supplied in 1.0 and in 1.1 with respect to untrusted classes and server sockets (either instances of class `ServerSocket` or datagram sockets that received data from any source). In 1.0, untrusted classes were typically not allowed to create a server socket at all, which meant that the `checkListen( )` and `checkAccept( )` methods always threw a security exception when an applet attempted such an operation. In 1.1 and later, untrusted classes are allowed to create a server socket so long as the port number of that socket is greater than the privileged port number on the machine (typically 1024). Note too that the `receive( )` method of the `DatagramSocket` class in Java 2 now calls the `checkAccept( )` rather than the `checkConnect( )` method.

Sun's 1.1−based browsers (HotJava and `appletviewer`) and some versions of Internet Explorer allow you to configure them so that untrusted classes can connect to any host on the network.

4.4.2.3 System access

In 1.1, attempts to redirect the standard input, output, and error streams call the `checkExec()` method rather than the `checkPermission( )` method. In fact, the `checkPermission( )` method does not exist at all in 1.1 and earlier releases.

4.4.2.4 Thread access

Thread access policy by `appletviewer` and popular browsers in 1.0 and 1.1 is, simply put, very confusing.

In 1.1, by default each applet is given an individual thread group, and the threads within that group can manipulate other threads within that group without respect to any additional hierarchy.

The `getThreadGroup( )` method is only present in Java 1.1 and subsequent releases. In Java 1.0 (and browsers built on that release), thread security was generally nonexistent: any thread could manipulate the state of any other thread, and applets weren't able to create their own thread groups.

4.4.2.5 Security access

1.1 implements all of the security checks that were listed earlier; in addition, the following methods also call the `checkSecurityAccess( )` method: `Identity.toString( )`, `Security.getProviders( )`, `Security.getProvider( )`, and `Security.getProperty( )`. However, since most browsers (including Netscape Communicator 4.x and Internet Explorer 4.x) do not implement the standard security package at all, none of these checks are performed in those browsers.

4.5 Summary

In this chapter, we've had an overview of the most commonly known feature of Java's security story: the security manager. The security manager is responsible for arbitrating access to what we normally consider operating system features −− files, network sockets, printers, etc. The goal of the security manager is to grant access to each class according to the amount of trust the user has in the class. Often, that means granting full access to trusted classes (that is, classes that have been loaded from the filesystem) while limiting access when the access is requested from an untrusted class (that is, a class that has been loaded from the network).

Although the security manager is the most commonly known feature of Java's security story, it's often misunderstood: there is no standard security manager among Java implementations, and Java applications, by default, have no security manager at all. Even with the popular Java−enabled browsers, the user often has latitude in what protections the security manager will be asked to enforce.

We examined in this chapter all the times when the security manager is asked to make a decision regarding access; such decisions range from the expected file and network access to more esoteric decisions, such as whether a frame needs a warning banner or what thread group a particular thread should belong to. This gave us a basic understanding of how the security manager is used to enforce a specific policy and the issues involved when defining such a policy. This knowledge will be used as a basis in the next few chapters when we'll look at how the security manager is generally implemented.

Chapter 5. The Access Controller

In the last chapter, we looked at the security manager, which provides the security policy interface used by the core Java API. The implementation of most security managers, however, is based entirely upon the access controller. In this chapter, we're going to look at the access controller and its related classes. Along the way, we'll cover a number of important topics:

- How to implement and use your own permission classes to extend the mechanism of the Java sandbox to your own applications.
- How to implement a different security policy so that permissions can be set in new ways (e.g., by reading them from a central server rather than from a collection of files).
- How the core Java API is able to perform certain operations that other classes cannot.
- How to create objects that can only be accessed if you have the appropriate permission.

The access controller is built upon the four concepts we examined in Chapter 2 :

Code sources
An encapsulation of the location from which certain Java classes were obtained.

Permissions
An encapsulation of a request to perform a particular operation.

Policies
An encapsulation of all the specific permissions that should be granted to specific code sources.

Protection domains
An encapsulation of a particular code source and the permissions granted to that code source.

We'll start by examining how these concepts map to the Java API.

5.1 The CodeSource Class

A code source is a simple object that reflects the URL from which a class was loaded and the keys (if any) that were used to sign that class. Class loaders are responsible for creating and manipulating code source objects, as we'll see in the next chapter.

The `CodeSource` class (`java.security.CodeSource`) has a few interesting methods:

public CodeSource(URL url, Certificate cers[])
Create a code source object for code that has been loaded from the specified URL. The optional array of certificates is the array of public keys that have signed the code that was loaded from this URL. These certificates are typically obtained from reading a signed jar file; if the code was not signed, this argument should be `null`. Similarly, the URL may be `null`.

public boolean equals(Object o)

Two code source objects are considered equal if they were loaded from the same URL (that is, the `equals( )` method for the URL of the objects returns `true`) and the array of certificates is equal (that is, a comparison of each certificate in the array of certificates will return `true`).

public final URL getLocation( )

Return the URL that was passed to the constructor of this object.

public final Certificate[] getCertificates( )

Return a copy of the array of certificates that was used to construct this code source object. The original certificates are not returned so that they cannot be modified accidentally (or maliciously).

public boolean implies(CodeSource cs)

Determine if the code source implies the parameter according to the rules of the `Permission` class (see later in this chapter). One code source implies another if it contains all the certificates of the parameter and if the URL of the parameter is implied by the URL of the target.

That's the extent of the `CodeSource` class. In Chapter 6, we'll show how to construct and use class loaders; later in this chapter, we'll see how they fit into the design of the access controller.

5.2 Permissions

The basic entity that the access controller operates on is a permission object -- an instance of the `Permission` class (`java.security.Permission`). This class, of course, is the basis of the types that are listed in a policy file for the default security policy. The `Permission` class itself is an abstract class that represents a particular operation. The nomenclature here is a little misleading because a permission object can reflect two things. When it is associated with a class (through a code source and a protection domain), a permission object represents an actual permission that has been granted to that class. Otherwise, a permission object allows us to ask if we have a specific permission.

For example, if we construct a permission object that represents access to a file, possession of that object does not mean we have permission to access the file. Rather, possession of the object allows us to ask if we have permission to access the file.

An instance of the `Permission` class represents one specific permission. A set of permissions -- e.g., all the permissions that are given to classes signed by a particular individual -- is represented by an instance of the `Permissions` class (`java.security.Permissions`). We've seen how administrators use the class in order to modify security policies; now we'll look into the programming behind this class, including defining your own permission classes.

5.2.1 The Permission Class

Recall that permissions have three properties: a type, a name, and some actions. The name and actions are optional. The type corresponds to an actual Java type (e.g., `java.io.FilePermission`); the types that are built into the core API are listed in Chapter 2.

The name of a permission is fairly arbitrary. In the case of file permissions, the name is obviously the file. But the name of the `showWindowWithoutWarningBanner` permission (among many others) is chosen by convention, and it is up to all Java programs to adhere to that convention. This is only a concern to programmers when dealing with your own permission classes; as a developer you rarely need to create permission objects for the types of permissions defined in the Java API. On the other hand, this naming convention is of concern to end users and administrators, who must know the name of the permission they want to grant to the programs they are going to run.

The presence of actions is dependent upon the semantics of the specific type of permission. A file permission object has a list of actions that could include read, write, and delete; an XYZ payroll permission object could have a list of actions that includes view and update. On the other hand, a window permission does not have an action: you either have permission to create the window or you don't. Actions can also be specified by wildcards. The terms used to specify a list of actions are also arbitrary and handled by convention.

Permissions can serve two roles. They allow the Java API to negotiate access to several resources (files, sockets, and so on). Those permissions are defined by convention within the Java API, and their naming conventions are wholly within the domain of the Java API itself. Hence, you can create an object that represents permission to read a particular file, but you cannot create an object that represents permission to copy a particular file since the copy action is not known within the file permission class.

On the other hand, you can create arbitrary permissions for use within your own programs and completely define both the names of those permissions as well as the actions (if any) that should apply. If you are writing a payroll program, for example, you could create your own permission class that uses the convention that the name of the permission is the employee upon whose payroll information you want to act; you could use the convention that the permissible actions on the payroll permission are view and update. Then you can use that permission in conjunction with the access controller to allow employees to view their own payroll data and to allow managers to change the payroll data for their employees.

From a developer perspective, this latter role is the more interesting, and we'll use it when we examine the permission class itself.

5.2.2 Using the Permission Class

We'll now look into the classes upon which all these permissions are based: the `Permission` class. This class abstracts the notion of a permission and a name. From a programmatic standpoint, the `Permission` class is really used only to create your own types of permissions. It has some interesting methods, but the operations that are implemented on a permission object are not generally used in code that we write -- they are used instead by the access controller. Hence, we'll examine this class primarily with an eye towards understanding how it can be used to implement our own permissions.

`Permission` is an abstract class that contains these public methods:

public Permission(String name)

Construct a permission object that represents the desired permission.

public abstract boolean equals(Object o)

Subclasses of the `Permission` class are required to implement their own test for equality. Often this is simply done by comparing the name (and actions, if applicable) of the permission.

public abstract int hashCode( )

Subclasses of the `Permission` class are required to implement their own hash code. In order for the access controller to function correctly, the hash code for a given permission object must never change during execution of the virtual machine. In addition, permissions that compare as equal must return the same hash code.

public final String getName( )

Return the name that was used to construct this permission.

public abstract String getActions( )

Return the canonical form of the actions (if any) that were used to construct this permission.

public String toString( )

The convention for printing a permission is to print in parentheses the class name, the name of the permission, and the actions. For example, a file permission might return:

```
("java.io.FilePermission","/myclasses/xyz/HRApplet.class","read")
```

public abstract boolean implies(Permission p)

This method is one of the keys of the `Permission` class: it is responsible for determining whether a class that is granted one permission is granted another. This method is normally responsible for performing wildcard matching so that, for example, the file permission */myclasses/−* implies the file permission */myclasses/xyz/HRApplet.class*. But this method need not rely on wildcards; permission to write a particular object in a database would probably imply permission to read that object as well.

public PermissionCollection newPermissionCollection( )

Return a permission collection suitable for holding instances of this type of permission. We'll discuss the topic of permission collections in the next section. This method returns `null` by default.

public void checkGuard(Object o)

Call the security manager to see if the permission (i.e., the `this` variable) has been granted, generating a `SecurityException` if the permission has not been granted. The object parameter of this method is unused. We'll give more details about this method later in this chapter.

Implementing your own permission means providing a class with concrete implementations of these abstract methods. Note that the notions of wildcard matching and actions are not generally present in this class −− if you want your class to support either of these features, you're responsible for implementing all of the necessary logic to do so (although the `BasicPermission` class that we'll look at next can help us with that).

Say that you are implementing a program to administer payroll information. You'll want to create permissions to allow users to view their payroll history. You'll also want to allow the HR department to update the pay rate for employees. We'll need to implement a permission class to encapsulate all of that:

```
package javasec.samples.ch05;

import java.security.*;
import java.util.*;

public class XYZPayrollPermission extends Permission {

    protected int mask;
```

```
static private int VIEW = 0x01;
static private int UPDATE = 0x02;

public XYZPayrollPermission(String name) {
    // Our permission must always have an action, so we
    // choose a default one here.
    this(name, "view");
}

public XYZPayrollPermission(String name, String action) {
    // Our superclass, however, does not support actions
    // so we don't provide one to that.
    super(name);
    parse(action);
}

private void parse(String action) {
    // Look in the action string for the words view and
    // update, separated by white space or by a comma
    StringTokenizer st = new StringTokenizer(action, ",\t ");

    mask = 0;
    while (st.hasMoreTokens(  )) {
        String tok = st.nextToken(  );
        if (tok.equals("view"))
            mask |= VIEW;
        else if (tok.equals("update"))
            mask |= UPDATE;
        else throw new IllegalArgumentException(
                                "Unknown action " + tok);
    }
}

public boolean implies(Permission permission) {
    if (!(permission instanceof XYZPayrollPermission))
        return false;

    XYZPayrollPermission p = (XYZPayrollPermission) permission;
    String name = getName(  );
    // The name must be either the wildcard *, which signifies
    // all possible names, or the name must match our name
    if (!name.equals("*") && !name.equals(p.getName(  )))
        return false;
    // Similarly, the requested actions must all match actions
    // that we've been constructed with.
    if ((mask & p.mask) != p.mask)
        return false;
    // Only if both the action and name match do we return true.
    return true;
}

public boolean equals(Object o) {
    if (!(o instanceof XYZPayrollPermission))
        return false;

    // For equality, we check the name and action mask.
    // We must provide a method definition like this, since
    // the security system expects us to do a deep check for
    // equality rather than relying on object reference
    // equality.
    XYZPayrollPermission p = (XYZPayrollPermission) o;
    return ((p.getName().equals(getName(  ))) && (p.mask == mask));
}
```

```
    public int hashCode(  ) {
        // We must always provide a hash code for permissions,
        // because the hashes must match if the permissions compare
        // as equals. The default implementation of this method
        // wouldn't provide that.
        return getName().hashCode(  ) ^ mask;
    }

    public String getActions(  ) {
        // This method must return the same string, no matter how
        // the action list was passed to the constructor.
        if (mask == 0)
            return "";
        else if (mask == VIEW)
            return "view";
        else if (mask == UPDATE)
            return "update";
        else if (mask == (VIEW | UPDATE))
            return "view, update";
        else throw new IllegalArgumentException("Unknown mask");
    }

    public PermissionCollection newPermissionsCollection(  ) {
        // More about this in a later example.
        return new XYZPayrollPermissionCollection(  );
    }

    public static void main(String[] args) {
        XYZPayrollPermission p1 =
                    new XYZPayrollPermission("sdo", "view");
        XYZPayrollPermission p2 =
                    new XYZPayrollPermission(args[0], args[1]);
        System.out.println("P1 is " + p1);
        System.out.println("P2 is " + p2);
        System.out.println("P1 -> P2 is " + p1.implies(p2));
        System.out.println("P2 -> P1 is " + p2.implies(p1));
    }
}
```

The instance variables in this class are required to hold the information about the actions −− even though our superclass makes references to actions, it doesn't provide a manner in which to store them or process them, so we have to provide that logic. That logic is provided in the `parse( )` method; we've chosen the common convention of having the action string treated as a list of actions that are separated by commas and whitespace. Note also that we've stored the actual actions as bits in a single integer −− this simplifies some of the later logic.

As required, we've implemented the `equals( )` and `hashCode( )` methods −− and we have done so rather simply. We consider objects equal if their names are equal and their masks (that is, their actions) are equal, and construct a hash code accordingly.

Our implementation of the `getActions( )` method is typical: we're required to return the same action string for a permission object that was constructed with an action list of `view, update` as for one that was constructed with an action list of `update, view`. This requirement is one of the prime reasons why the actions are stored as a mask −− because it allows us to construct this action string in the proper format.

Finally, the `implies( )` method is responsible for determining how wildcard and other implied permissions are handled. If the name passed to construct our object is an asterisk, then we match any other name; hence, an object to represent the permissions of the HR department might be constructed as:

```
new XYZPayrollPermission("*", "view, update")
```

When the `implies( )` method is called on this wildcard object, the name will always match, and because the action mask has the complete list of actions, the mask comparison will always yield the mask that we're testing against. If the `implies( )` method is called with a different object, however, it will only return `true` if the names are equal and the object's mask is a subset of the target mask.

Note that we also might have implemented the logic in such a way that permission to perform an update implies permission to perform a view simply by changing the logic of testing the mask −− you're not limited only to wildcard matching in the `implies( )` method.

5.2.3 The BasicPermission Class

If you need to implement your own permission class, the `BasicPermission` class (`java.security.BasicPermission`) provides some useful semantics. This class implements a basic permission −− that is, a permission that doesn't have actions. Basic permissions can be thought of as binary permissions −− you either have them or you don't. However, this restriction does not prevent you from implementing actions in your subclasses of the `BasicPermission` class (as the `PropertyPermission` class does).

The prime benefit of this class is the manner in which it implements wildcards. Names in basic permissions are considered to be hierarchical, following a dot−separated convention. For example, if the XYZ corporation wanted to create a set of basic permissions, they might use the convention that the first word of the permission always be `xyz`: `xyz.readDatabase`, `xyz.writeDatabase`, `xyz.runPayrollProgram`, `xyz.HRDepartment.accessCheck`, and so on. These permissions can then be specified by their full name, or they can be specified with an asterisk wildcard: `xyz.*` would match each of these (no matter what depth), and `*` would match every possible basic permission.

The wildcard matching of this class does not match partial names: `xyz.read*` wouldn't match any of the permissions we just listed. Further, the wildcard must be in the rightmost position: `*.readDatabase` wouldn't match any basic permission.

The `BasicPermission` class is abstract, although it does not contain any abstract methods, and it completely implements all the abstract methods of the `Permission` class. Hence, a concrete implementation of the `BasicPermission` need only contain a constructor to call the correct constructor of the superclass (since there is no default constructor in the `BasicPermission` class). Subclasses must call one of these constructors:

public BasicPermission(String name)
: Construct a permission with the given name. This is the usual constructor for this class, as basic permissions do not normally have actions.

public BasicPermission(String name, String action)
: Construct a permission with the given name and action. Even though basic permissions do not usually have actions associated with them, you must provide a constructor with this signature in all implementations of the `BasicPermission` class due to the mechanism that is used to construct permission objects from the policy file (which we will see later in this chapter).

5.2.4 Permission Collections

The access controller depends upon the ability to aggregate permissions so that it can easily call the `implies( )` method on all of them. For example, a particular user might be given permission to read several directories: perhaps the user's home directory (*/home/sdo/−*) and the system's temporary directory (*/tmp/−*). When the access controller needs to see if the user can access a particular file, it must test both of these permissions to see if either one matches. This can be done easily by aggregating all the file permissions into a single permission collection.

Every permission class is required to implement a permission collection, then, which is a mechanism where objects of the same permission class may be grouped together and operated upon as a single unit. This requirement is enforced by the `newPermissionCollection( )` method of the `Permission` class.

The `PermissionCollection` class (`java.security.PermissionCollection`) is defined as follows:

public abstract class PermissionCollection

Implement an aggregate set of permissions. While permission collections can handle heterogeneous sets of permissions, a permission collection typically should be used to group together a homogeneous group of permissions (e.g., all file permissions or all socket permissions, etc.).

There are three basic operations that you can perform on a permission collection:

public abstract void add(Permission p)

Add the given permission to the permission collection.

public abstract boolean implies(Permission p)

Check to see if any permission in the collection implies the given permission. This can be done by enumerating all the permission objects that have been added to the collection and calling the `implies( )` method on each of those objects in turn, but it is typically implemented in a more efficient manner.

public abstract Enumeration elements( )

Return an enumeration of all the permissions in the collection.

The definition of this class seems to imply that permission collections can contain any set of arbitrary permissions (and some versions of the Java documentation state that explicitly). Forget that idea; introducing that notion into permission collections vastly complicates matters, and the issue of a heterogeneous collection of permission objects is better handled elsewhere (we'll see how a little bit later). As far as we're concerned, the purpose of a permission collection is to aggregate only permission objects of a particular type.

Permission collections are typically implemented as inner classes, or at least as classes that are private to the package in which they are defined. There is, for example, a corresponding permission collection class for the `FilePermission` class, one for the `SocketPermission` class, and so on.

None of these collections is available as a public class that we can use in our own program. Hence, in order to support the `newPermissionCollection( )` method in our `XYZPayrollPermission` class, we'd need to do something like this:

```
package javasec.samples.ch05;

import java.util.*;
import java.security.*;
import java.util.*;
```

```
public class XYZPayrollPermissionCollection
                              extends PermissionCollection {
    private Hashtable permissions;
    // We keep track of whether the * name has been added to make
    // the implies method simpler.
    private boolean addedAdmin;
    private int adminMask;

    XYZPayrollPermissionCollection(  ) {
        permissions = new Hashtable(  );
        addedAdmin = false;
    }

    public void add(Permission p) {
        // Required test
        if (isReadOnly(  ))
            throw new IllegalArgumentException("Read only collection");

        // This is a homogenous collection, as are all
        // PermissionCollections that you'll implement.
        if (!(p instanceof XYZPayrollPermission))
            throw new IllegalArgumentException("Wrong type");

        XYZPayrollPermission xyz = (XYZPayrollPermission) p;
        String name = xyz.getName(  );
        XYZPayrollPermission other =
                       (XYZPayrollPermission) permissions.get(name);
        if (other != null)
            xyz = merge(xyz, other);
        // An administrative permission. The administrative permission
        // may have only view or only update or both, and multiple
        // admin permissions may be added, so the masks are OR−ed
        // together.
        if (name.equals("*")) {
            addedAdmin = true;
            adminMask = xyz.mask | adminMask;
        }
        permissions.put(name, xyz);
    }

    public Enumeration elements(  ) {
        return permissions.elements(  );
    }

    public boolean implies(Permission p) {
        if (!(p instanceof XYZPayrollPermission))
            return false;
        XYZPayrollPermission xyz = (XYZPayrollPermission) p;
        // If the admin name is present, then all names are implied;
        // we need check only the permissions.
        if (addedAdmin && (adminMask & xyz.mask) == xyz.mask)
            return true;
        // Otherwise, we can just see if the given individual is
        // in our table and if so, see if the permissions match.
        Permission inTable =
                  (Permission) permissions.get(xyz.getName(  ));
        if (inTable == null)
            return false;
        return inTable.implies(xyz);
    }

    // This is called when the same name is added twice to the
    // permissions; we merge the action lists and only store the
    // name once.
    private XYZPayrollPermission merge(XYZPayrollPermission a,
                                        XYZPayrollPermission b) {
        String aAction = a.getActions(  );
        if (aAction.equals(""))
            return b;
        String bAction = b.getActions(  );
        if (bAction.equals(""))
            return a;
        return new XYZPayrollPermission(a.getName(  ),
                              aAction + "," + bAction);
    }
}
```

Note the logic within the `implies( )` method −− it's the important part of this example. The `implies( )` method must test each permission in the hashtable (or whatever other container you've used to store the added permissions), but it should do so efficiently. We could always call the `implies( )` method of each entry in the hashtable, but that would clearly not be efficient −− it's better to call only the `implies( )` method on a permission in the table that has a matching name.

The only trick is that we won't find a matching name if we're doing wildcard pattern matching −− if we've added the name "*" to the table, we'll always want to return `true`, even though looking up the name "John Smith" in the table will not return the administrative entry. Implementing this wildcard pattern matching efficiently is the key to writing a good permission collection.

When you use (or subclass) one of the concrete permission classes that we listed in Chapter 2, there is no need to provide a permission collection class −− all concrete implementations provide their own collection. In addition, there are two other cases when you do not need to implement a permission collection:

- When you extend the `Permission` class, but do not do wildcard pattern matching.

  Hidden internally within the Java API is a `PermissionsHash` class, which is the default permission collection class for permission objects. The `PermissionsHash` class stores the aggregated permissions in a hashtable, so the implementations of its `add( )` and `elements( )` methods are straightforward. The implementation of its `implies( )` method is based on looking up the name of the permission parameter in the hashtable collection: if an entry is found, then the `implies( )` method is called on that entry.
- When you extend the `BasicPermission` class and do not provide support for actions.

  The `newPermissionClass( )` method of the `BasicPermission` class will provide a permission collection that handles wildcard pattern matching correctly (and efficiently).

If you implement your own `PermissionCollection` class, you must keep track of whether it has been marked as read−only. There are two methods involved in this:

public boolean isReadOnly( )
: Return an indication of whether the collection has been marked as read−only.

public void setReadOnly( )
: Set the collection to be read−only. Once the read−only flag has been set, it cannot be cleared: the collection will remain read−only forever.

A permission collection is expected to throw a security exception from its `add( )` method if it has been marked as read−only. Note the read−only instance variable is private to the `PermissionCollection` class, so subclasses will have to rely on the `isReadOnly( )` method to test its value.

5.2.5 The Permissions Class

So far, we've described permission collections as homogeneous collections: all permissions in the `XYZPayrollPermissionCollection` class are instances of the `XYZPayrollPermission` class; a similar property holds for other permission collections. This idea simplifies the `implies( )` method that we showed previously. But to be truly useful, a permission collection needs to be heterogeneous so that it can represent all the permissions a program should have. A permission collection really needs to be able to contain file permissions, socket permissions, and other types of permissions.

This idea is present within the `PermissionCollection` class; conceptually, however, it is best to think of heterogeneous collections of permissions as encapsulated by the `Permissions` class (`java.security.Permissions`):

public final class Permissions extends PermissionCollection

Implement the `PermissionCollection` class. This class allows you to create a heterogeneous collection of permissions: the permission objects that are added to this collection need not have the same type.

This class contains a concrete implementation of a permission collection that organizes the aggregated permissions in terms of their individual, homogenous permission collections. You can think of a permissions object as containing an aggregation of permission collections, each of which contains an aggregation of individual permissions.

As an example, let's consider an empty permissions object. When a file permission is added to this object, the permissions object will call the `newPermissionCollection( )` method on the file permission to get a homogeneous file permission collection object. The file permission is then stored within this file permission collection. When another file permission is added to the permissions object, the permissions object will place that file permission into the already existing file permission collection object. When a payroll permission object is added to the permissions object, a new payroll permission collection will be obtained, the payroll permission added to it, and the collection added to the permissions object. This process will continue, and the permissions object will build up a set of permission collections.

When the `implies( )` method of the permissions object is called, it will search its set of permission collections for a collection that can hold the given permission. It can then call the `implies( )` method on that (homogenous) collection to obtain the correct answer.

The `Permissions` class thus supports any arbitrary grouping of permissions. There is no need to develop your own permission collection to handle heterogeneous groups.

5.3 The Policy Class

The third building block for the access controller is the facility to specify which permissions should apply to which code sources. We call this global set of permissions the security policy; it is encapsulated by the `Policy` class (`java.security.Policy`).

public abstract class Policy

Establish the security policy for a Java program. The policy encapsulates a mapping between code sources and permission objects in such a way that classes loaded from particular locations or signed by specific individuals have the set of specified permissions.

A policy class is constructed as follows:

public Policy( )

Create a policy class. The constructor should initialize the policy object according to its internal rules (e.g., by reading the *java.policy* file).

There are two other methods in the `Policy` class:

public abstract Permissions getPermissions(CodeSource cs)

Create a permissions object that contains the set of permissions that should be granted to classes that came from the given code source (i.e., loaded from the code source's URL and signed by the keys in the code source).

public abstract void refresh( )

Refresh the policy object. For example, if the initial policy came from a file, re−read the file and install a new policy object based on the (presumably changed) information from the file.

In programmatic terms, writing a policy class involves implementing these methods. The default policy class is provided by the `PolicyFile` class (`sun.security.provider.PolicyFile`), which constructs permissions based on information found in the appropriate policy files.

Unfortunately, the `PolicyFile` class is in the `sun` package; it is not accessible to us as programmers. Hence, while it's possible to write your own `Policy` class, it is a fairly involved process. You might want to write your own `Policy` class if you want to define a set of permissions through some other mechanism than a URL (e.g., loading the permissions via a policy server database). That implementation is fairly straightforward: you need only provide a mechanism to map code sources to a set of permissions. Then, for each code source, construct each of the individual permission objects and aggregate them into a permissions object to be returned by the `getPermissions( )` method.

Hence, a simple implementation of a policy class might looks like this:

```
package javasec.samples.ch05;

import java.security.*;
import java.util.*;

public class MyPolicy extends Policy {

    // This inner class defines a simple set of permissions:
    // either everything is allowed (the implies(  ) method always
    // returns true, and the collection contains an AllPermission
    // object) or everything is prohibited (the implies(  )
    // method always returns false and the collection is empty).
    static class SimplePermissions extends PermissionCollection {
        boolean allow;
        Permissions perms;

        SimplePermissions(boolean b) {
            allow = b;
            perms = new Permissions(  );
            if (allow)
                perms.add(new AllPermission(  ));
        }

        public void add(Permission p) {
            if (isReadOnly(  ))
                throw new SecurityException(
                             "Can't add to this collection");
            perms.add(p);
        }

        public Enumeration elements(  ) {
            return perms.elements(  );
        }

        public boolean implies(Permission p) {
            return allow;
        }
    }

    // We never change the policy
    public void refresh(  ) {}

    // If the code was loaded from a file, return a collection
    // that implies all permissions. Otherwise, return an
    // empty collection (one that implies no permissions).
    public PermissionCollection getPermissions(CodeSource cs) {
        if (cs.getLocation().getProtocol(  ).equals("file"))
            return new SimplePermissions(true);
        return new SimplePermissions(false);
    }
}
```

This class implements a simple policy: if the class in question was loaded from the filesystem, then all operations are allowed. Otherwise, all operations are denied. Note that this implementation requires us to define a permission collection to return from the `getPermissions(  )` method. That's typically where all the work goes; in our case, we have either a simple collection with an all permission element or one with no elements. The `implies(  )` method in this case doesn't rely upon those elements, though it typically would.

5.3.1 Installing a Policy Class

Only a single instance of the policy class can be installed in the virtual machine at any time. However, the actual instance of the policy class can be replaced. There is a programmatic and an administrative way to install a policy class. As it turns out, both ways have their problems, though the administrative way is preferable.

These two methods install and retrieve the policy programatically:

public static Policy getPolicy( )

Return the currently installed policy object.

public static void setPolicy(Policy p)

Install the given policy object, replacing whatever policy object was previously installed.

Getting and setting the policy object requires going through the `checkProperty(  )` method of the security manager. By default, this succeeds only if you already have been granted a security permission with the name of `getPolicy` or `setPolicy` (as appropriate).

When the new policy is installed, it will only affect code sources that have not yet been established. To use this technique, you must install a policy file and then load classes through a new class loader; only those classes will be subject to the policy file that you've just installed.

You may also specify administratively which policy class to use by changing the `policy.provider` entry in the *java.security* file. By default, that entry is:

```
policy.provider=sun.security.provider.PolicyFile
```

So you can change `sun.security.provider.PolicyFile` to `javasec.samples.ch05.MyPolicy`, and that will be the default policy. But the `MyPolicy` class must reside in the boot classpath of the virtual machine. That means that in order to use this technique, you must either add `MyPolicy.class` to the *rt.jar* file, or you must run the virtual machine with the appropriate argument to specify the boot classpath. This argument is nonstandard and is subject to change, but in 1.3 to load the `MyPolicy` class from */files/policy* you'd use this command fragment:

```
piccolo% java -Xbootclasspath:/files/policy  ...other args...
```

Nonetheless, installing the policy file through the *java.security* file is far easier on developers since it will apply to all classes, regardless of their class loader.

5.4 Protection Domains

A protection domain is a grouping of a code source and permissions −− that is, a protection domain represents all the permissions that are granted to a particular code source. In terms of the policy files that we looked at in Chapter 2, a protection domain is one grant entry. A protection domain is an instance of the `ProtectionDomain` class (`java.security.ProtectionDomain`) and is constructed as follows:

public ProtectionDomain(CodeSource cs, PermissionCollection p)
  Construct a protection domain based on the given code source and set of permissions.

When associated with a class, a protection domain means that the given class was loaded from the site specified in the code source, was signed by the public keys specified in the code source, and should have permission to perform the set of operations represented in the permission collection object. Each class in the virtual machine may belong to one and only one protection domain, which is set by the class loader when the class is defined.

However, not all class loaders have a specific protection domain associated with them: the class loader that loads the core Java API does not specify a protection domain. We can think of these core classes as belonging to the system protection domain.

There are three utility methods of the `ProtectionDomain` class:

public CodeSource getCodeSource( )
  Return the code source that was used to construct this protection domain.

public PermissionCollection getPermissions( )
  Return the permission collection object that was used to construct this protection domain.

public boolean implies(Permission p)

Indicate whether the given permission is implied by the permissions object contained in this protection domain.

Although protection domain objects are important to the access controller, for most developers they are simply opaque objects.

5.5 The AccessController Class

Now we have all the pieces in place to discuss the mechanics of the access controller. The access controller is represented by a single class called, conveniently, `AccessController` (`java.security.AccessController`). There are no instances of the `AccessController` class −− its constructor is private so that it cannot be instantiated. Instead, this class has a number of static methods that can be called in order to determine if a particular operation should succeed. The key method of this class takes a particular permission and determines, based on the installed `Policy` object, whether or not the permission should be granted:

public static void checkPermission(Permission p)

Check the given permission against the policy in place for the program. If the permission is granted, this method returns normally; otherwise, it throws an `AccessControlException`.

This method is used by the security manager to implement each of its methods.

We can use this method to determine whether or not a specified operation should be permitted:

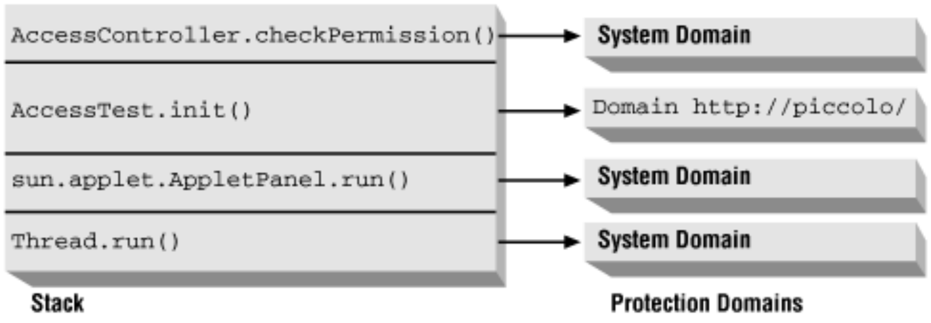
```
package javasec.samples.ch05;

import java.applet.*;
import java.net.*;
import java.security.*;

public class AccessTest extends Applet {
    public void init(  ) {
        SocketPermission sp = new SocketPermission(
                            getParameter("host") + ":6000", "connect");
        try {
            AccessController.checkPermission(sp);
            System.out.println("Ok to open socket");
        } catch (AccessControlException ace) {
            System.out.println(ace);
        }
    }
}
```

Whether the access controller allows or rejects a given permission depends upon the set of protection domains that are on the stack when the access controller is called. Figure 5−1 shows the stack that might be in place when the `init(  )` method of the `AccessTest` applet is called. In the `appletviewer`, an applet is run in a separate thread −− so the bottom method on the stack is the `run(  )` method of the `Thread` class.[1] That `run(  )` method has called the `run(  )` method of the `AppletPanel` class. This second `run(  )` method has done several things prior to calling the `init(  )` method: it first created an HTTP−based class loader (from an internal class that is a subclass of the `URLClassLoader` class) and has used that class loader to load the `AccessTest` class. It then instantiated an instance of the `AccessTest` class and called the `init(  )` method on that object. This left us with the stack shown in the figure −− the `run(  )` method of the `Thread` class has called the `run(  )` method of the `AppletPanel` class, which has called the `init(`

`)` method of the `AccessTest` class, which has called the `checkPermission( )` method of the `AccessController` class.

[1] The `run( )` method is always the bottom method on a stack since stacks apply on a per−thread basis.

Figure 5−1. The stack and protection domains of a method



The reason we need to know the stack trace of the current thread is to examine the protection domains that are on the stack. In Chapter 2, we referred to the classes on the stack as the active classes; the access controller looks at only those classes and domains that are on the stack.

In this example, only the `AccessTest` class has been loaded by a class loader: the `AppletPanel` class and the `Thread` class were loaded from the core API with the primordial class loader. Hence, only the `AccessTest` class has a nonsystem protection domain (associated with the URL from which we loaded it, *http://piccolo/* in this case).

The permissions in effect for any particular operation are the intersection of all permissions of each protection domain on the stack when the `checkPermission( )` method is called. When the `checkPermission( )` method is called, it checks the permissions associated with the protection domain for each method on the stack. It does this starting at the top of the stack and proceeding through each class.

If this entry appeared in the policy file:

```
grant codeBase http://piccolo/ {
        permission java.net.SocketPermission "*:1024−", "connect";
};
```

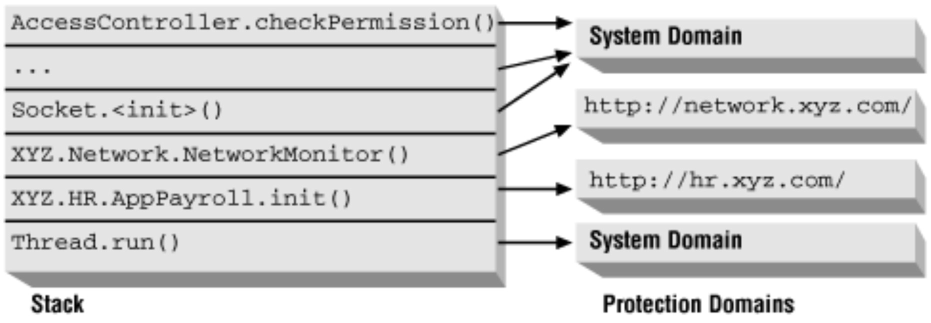
the protection domain that applies to the `AccessTest` class will have permission to open the socket. Remember that the system domain implicitly has permission to perform any operation; as there are no other nonsystem protection domains associated with any class on the stack, the `checkPermission( )` method will permit this operation −− which is to say that it will silently return.

For most implementations of Java browsers, and many Java applications, there will only be a single nonsystem protection domain on the stack: all the classes for the applet will have come from a single `CODEBASE` (and hence a single protection domain). But the `checkPermission( )` method is more general than that, and if you use a class loader that performs delegation, there will be multiple protection domains on the stack. This is a common occurrence if you're using a Java extension or your own custom class loader.

Let's say that you've written a payroll application that uses a class loader that loads classes from two sources: the server in the XYZ HR department and the server in the XYZ network services department.[2] This might lead to a call to the `checkPermission( )` method with the stack shown in Figure 5−2. Note that this stack trace is a little more complicated than the one we've just shown −− in this case, we're relying on the fact that the constructor of the `Socket` class indirectly calls the access controller (because, as we mentioned in Chapter 4, it eventually calls the security manager, which calls the access controller).

[2] We'll show this example and class loaders to implement it in Chapter 6.

Figure 5−2. A stack with multiple nonsystem protection domains



In this example, the access controller first checks the protection domain for the `Network` class to see if a class loaded from *http://network.xyz.com/* is allowed to connect to the socket. If that succeeds, it then checks the protection domain of the `PayrollApp` class to see if a class loaded from *http://hr.xyz.com/* is allowed to connect to the socket. Only if both code sources are granted permission in the policy file (either individually or via an entry that does not specify a code base at all) does the `checkPermission( )` method succeed.

Whether or not this is the appropriate behavior depends upon your intent. Let's say that the policy file for the payroll application specifies that classes with a code base of *http://network.xyz.com/* are allowed to create sockets but that no other protection domains (other than the system protection domain, of course) are granted that permission. That leads to the situation where a class from the network services department might not be able to open a socket (even though it has that permission in the file): if there is any class in the HR protection domain on the stack, the operation will fail. All classes on the stack must have permission for an operation to succeed.

Often, however, you want a class to be temporarily given the ability to perform an action on behalf of a class that might not normally have that ability. In this case, we might want to establish a policy where the classes from the HR department cannot create a socket directly but where they can call classes from the network services department that can create a socket.[3] In this case, you want to tell the access controller to grant (temporarily) the permissions of the network services department to any methods that it might call within the current thread.

[3] Consider this in terms of writing a file: an applet might not be able to write a file, but it can call a method of the SDK to play audio data −− which means the SDK class must write to the audio device file.

That facility is possible with these methods of the access controller class:

public static Object doPrivileged(PrivilegedAction pa)
public static Object doPrivileged(PrivilegedExceptionAction pae)
public static Object doPrivileged(PrivilegedExceptionAction action, AccessControlContext context)

Execute the `run( )` method on any given object, temporarily granting its permission to any protection domains below it on the stack. In the second case, if the embedded `run( )` method throws an exception, the `doPrivileged( )` method will throw a `PrivilegedActionException`.

The `PrivilegedAction` and `PrivilegedExceptionAction` interfaces contain a single method:

public Object run( )

Run the target code, which will have the permissions of the calling class.

The difference between the two interfaces is that the `run( )` method in the `PrivilegedExceptionAction` interface may throw an arbitrary exception. Note the unfortunate overloading between this method and the `run( )` method of the `Thread` class and `Runnable` interface, which return `void`; a class cannot implement both the `Runnable` and `PrivilegedAction` interfaces.

The `PrivilegedActionException` class is a standard exception, so you must always be prepared to catch it when using the `doPrivileged( )` method. If the embedded `run( )` method does throw an exception, that exception will be wrapped into the `PrivilegedActionException`, where it may be retrieved with this call:

public Exception getException( )

Return the exception that was originally thrown to cause the `PrivilegedActionException` to be thrown.

Let's see how all of this might work with our network monitor example:

```
package javasec.samples.ch05;

import java.net.*;
import java.io.*;
import java.security.*;

public class NetworkMonitor {
    public NetworkMonitor(  ) {
        try {
            // This class is used by the doPrivileged(  ) method to
            // open a socket
            class doSocket implements PrivilegedExceptionAction {
                public Object run(  ) throws UnknownHostException,
                                     IOException {
                    return new Socket("net.xyz.com", 4000);
                }
            };
            doSocket ds = new doSocket(  );
            Socket s = (Socket) AccessController.doPrivileged(ds);
        } catch (PrivilegedActionException pae) {
            Exception e = pae.getException(  );
            if (e instanceof UnknownHostException) {
                // process host exception
            }
            else if (e instanceof IOException) {
                // process IOException
            }
            else {
                // e must be a runtime exception
                throw (RuntimeException) e;
            }
        }
    }
}
```

Two points are noteworthy here. First, the code that needs to be executed with the privileges of the `NetworkMonitor` class has been encapsulated into a new class -- the inner `doSocket` class.

Second, the exception handling is somewhat new: we must list the exceptions that the socket constructor can throw in the `run( )` method of our embedded class. If either of those exceptions is thrown, it will be encapsulated into a `PrivilegedActionException` and thrown back to the network monitor, where we can retrieve the actual exception with the `getException( )` method.

Let's examine the effect this call has on the access controller. The access controller begins the same way, by examining the protection domains associated with each method on the stack. But this time, rather than searching every class on the stack, the access controller stops searching the stack when it reaches the class that has called the `doPrivileged( )` method. In the case of Figure 5−2, this means that the access controller does not continue searching the stack after the `NetworkMonitor` class, so as long as the policy file has a valid entry for the *http://network.xyz.com/* codebase, the monitor will be able to create its socket.

There's an important (but subtle) distinction to be made here: the `doPrivileged( )` method does not establish a global permission based on the protection domain of the class that called it. Rather, it specifies a stopping point as the access controller searches the list of protection domains on the stack. In the previous example, we assumed that *http://network.xyz.com/* had permission to open the socket. When the access controller searched the protection domains on the stack, it first reached the protection domain associated with *http://network.xyz.com/*. Since that domain had been marked as the privileged domain, the access controller returned at that point: it never got to the point on the stack where it would have checked (and rejected) the protection domain associated with *http://hr.xyz.com/*.

Now consider what would happen if the permissions given to these protection domains were reversed; that is, if the *http://network.xyz.com/* protection domain is not given permission to open the socket, but the *http://hr.xyz.com/* protection domain is. When we need to write a `PayrollApp` class, we might assume that the class has permission to open the socket and write it like this:

```
package javasec.samples.ch05;

import java.security.*;

public class PayrollApp {
    NetworkMonitor nm;
    public void init(  ) {
        class doInit implements PrivilegedAction {
            public Object run(  ) {
                nm = new NetworkMonitor(  );
                return nm;
            }
        }
        doInit di = new doInit(  );
        AccessController.doPrivileged(di);
    }
}
```

When the code within the `Socket` constructor calls the `checkPermission( )` method, the access controller searches the same stack shown in Figure 5−2. When the access controller reaches the protection domain associated with *http://network.xyz.com*, it immediately throws an `AccessControlException` because that protection domain does not have permission to open sockets. Even though a protection domain lower in the stack does have such a permission and even though that protection domain has called the `doPrivileged( )` method of the access controller, the operation is rejected when the access controller finds a protection domain that does not have the correct permission assigned to it.

So protection domain can grant privileges to code that has called it, but it cannot grant privileges to code that it calls. This rule permits key operations of the Java virtual machine; if, for example, your nonprivileged class calls the Java API to play an audio clip, the Java API will grant permission to the calling code to write data to the audio device on the machine. When you write your own applications, however, it's important to realize that the permission granting goes only one way.

5.5.1 Access Control Contexts

The access controller works by examining the protection domains of every class on the stack. Sometimes, however, you want the access controller to examine the protection domains of a different set of classes. You do that by using the `AccessControlContext` class (`java.security.AccessControlContext`):

public final class AccessControlContext
: Build a context based on a set of protection domains.

Instances of this class are returned by the `getContext( )` method of the `AccessController` class. That method takes a snapshot of the protection domains that are on the stack when it is called and stores those domains into the returned access context object.

You can then use that context object in a later call to the access controller; the access controller will use that set of protection domains in its operation rather than the set of domains that are on the stack.

Usage of this class in application code is very limited. It is often used within a class loader because the permissions that a class loader should use to determine whether a new class can be loaded should be determined by the protection domains of the stack that created the class loader rather than any subsequent users of that class loader.

5.6 Guarded Objects

The notion of permissions and the access controller can be encapsulated into a single object: a guarded object, which is implemented by the `GuardedObject` class (`java.security.GuardedObject`). This class allows you to embed another object within it in such a way that all access to the object will first have to go through a guard (which, typically, is the access controller).

There are two methods in the `GuardedObject` class:

public GuardedObject(Object o, Guard g)
: Create a guarded object. The given object is embedded within the guarded object; access to the embedded object will not be granted unless the guard allows it.

public Object getObject( )
: Return the embedded object. The `checkGuard( )` method of the guard is first called; if the guard prohibits access to the embedded object, an `AccessControlException` will be thrown. Otherwise, the embedded object is returned.

The guard can be any class that implements the `Guard` interface (`java.security.Guard`). This interface has a single method:

public void checkGuard(Object o)
: See if access to the given object should be granted. If access is not granted, this method should throw an `AccessControlException`; otherwise it should silently return.

Although you can write your own guards, the `Permission` class already implements the guard interface. Hence, any permission can be used to guard an object as follows:

```
package javasec.samples.ch05;

import java.security.*;

// This is a dummy class; it would usually be the object
// that we want to protect access to (e.g., a real payroll
// request would fall into that category).
class XYZPayrollRequest {}

public class GuardTest {
    public static void main(String args[]) {
        // Protect the given object by requiring the given
        // permission.
        GuardedObject go = new GuardedObject(new XYZPayrollRequest(  ),
                         new XYZPayrollPermission("sdo", "view"));
        try {
            Object o = go.getObject(  );
            System.out.println("Got access to object");
        } catch (AccessControlException ace) {
            System.out.println("Can't access object");
        }
    }
}
```

When the `getObject( )` method is called, it in turn calls the `checkGuard( )` method of the `XYZPayrollPermission` class, which (as it inherits from the `Permission` class) will call the `checkPermission( )` method of the access controller, passing the XYZ payroll permission object as an argument. Hence, in order to get access to the object, the necessary policy file must be created with the following permission granted to the codebase from which the `GuardTest` class is loaded:

```
permission javasec.samples.ch05.XYZPermission "sdo", "view";
```

A more permissive name and set of actions may be given as well.

5.7 Comparison with Previous Releases

There are no changes in the access controller and its associated methods and classes between Java 2 versions 1.2 and 1.3. None of the classes that we've discussed in this chapter are available in Java 1.1; in order to implement something similar, you would need to supply the logic we have discussed within a custom security manager.

5.8 Summary

In this chapter, we've looked at Java's access control mechanism. The access controller is the most powerful security feature of the Java platform: it protects most of the vital resources on a user's machine, and it allows users (or system administrators) to customize the security policy of a particular application simply by modifying entries in *java.policy* and other similar files.

The access controller is able to control access to a well−established set of system resources (files, sockets, etc.), but it is extensible as well: you can create permission classes that the access controller can use to grant or deny access to any resource that you like.

In the next chapter, we'll look how the class loader completes the implementation of a security policy by associated code sources and protection domains with specific classes.

Chapter 6. Java Class Loaders

In this chapter, we're going to look at the third major component that determines the security policy of a Java program: the Java class loader. Class loaders are the mechanism by which files (or other sources) containing Java bytecodes are read into the Java virtual machine and converted into class definitions.

There are three areas in which the class loader operates with the security model. First, the class loader cooperates with the virtual machine to define namespaces, which protect the integrity of the security features built into the Java language. Second, the class loader calls the security manager when appropriate, ensuring that code has the appropriate permissions in order to access or define classes. And third, the class loader sets up the mapping of permissions to class objects (the protection domain of each class) so that the access controller knows which classes have which permissions. The last of these areas is the one which is of most use to developers: if you want to establish a different security policy in your application, it's easier to do it by writing a custom class loader and establishing the permissions of classes within that class loader than by writing a new implementation of the `Policy` class.

In this chapter, we'll address all of these points. We'll also look into the class loader classes that come with Java and how to write your own class loader. As with the other elements of the Java sandbox, the ability to create and use a class loader is limited. If the default security model is in place, then you must explicitly grant code permission to create a class loader.

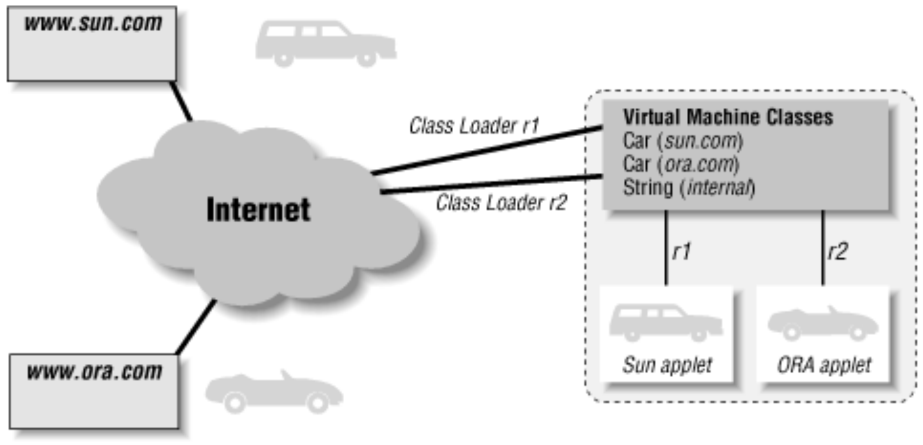
6.1 The Class Loader and Namespaces

Class loaders are used by the Java virtual machine to enforce certain rules about the namespaces used by Java classes. Recall that the full name of a Java class is qualified by the name of the package to which the class belongs; there is no standard class called `String` in the Java API, but there is the class `java.lang.String`. On the other hand, a class does not need to belong to a package, in which case its full name is just the name of the class. It's often said that these classes are in the default package, but that's slightly misleading: as it turns out, there is a different default package for each class loader in use by the virtual machine.

Consider what happens if you surf to a page at *www.sun.com* and load an applet that uses a class called `Car` (with no package name); after that, you surf to a page at *www.ora.com* and load a different applet that uses a class called `Car` (also with no package name). Clearly, these are two different classes, but they have the same fully qualified name −− how can the virtual machine distinguish between these two classes?

The answer to that question lies in the internal workings of the class loader. When a class is loaded by a class loader, it is stored in a reference internal to that class loader. A class loader in Java is simply an object whose type extends the `ClassLoader` class. When the virtual machine needs access to a particular class, it asks the appropriate class loader. For example, when the virtual machine is executing the code from *sun.com* and needs access to the `Car` class, it asks the class loader that loaded the applet (`r1` in Figure 6−1) to provide that class.

Figure 6−1. Different instances of the class loaders help to disambiguate class names



In order for this scheme to work, the `Car` class from *www.ora.com* must be loaded using a different class loader than that which loaded the `Car` class from *www.sun.com*. That way, when the virtual machine asks the class loader `r2` for the definition of the `Car` class, it will get back (correctly) the definition from *ora.com*. The class loader does not need to be a different class; as this example implies, it must merely be a different *instance* of the class. Hence, applets that have a different codebase (even if they originate on the same host) are always loaded by different instances of the browser's class loader. Applets on the same page with the same codebase, however, may use the same class loader so that they may share class files (as well as sharing other information). Some browsers also allow applets on different pages to be loaded by the same class loader as long as those applets have the same codebase, which is generally a more efficient and useful implementation.

This differentiation between class files loaded from different class loaders occurs no matter what packages are involved. Don't be confused by the fact that there were no explicit package names given in our example. A large computer company might define a class named `com.sun.Car`, a large oil company might also define a class called `com.sun.Car`, and the two classes need to be considered as distinct classes −− which they will be if (and only if) they are loaded by different instances of the class loader.

So far we've given a logical reason why the class loader is involved in the namespace resolution of Java classes. You might think that if everyone were to follow the convention that the beginning of their package name must be their Internet domain in reverse order −− e.g., `com.sun` for Sun Microsystems −− this idea of different class loaders wouldn't be necessary. But there are security reasons for this namespace separation as well.

In Java, classes that are members of the same package have certain privileges that other classes do not have −− they can access all the classes of the package that have the default protection (that is, the classes that are neither `public`, `private`, nor `protected`). Additionally, they can access any instance variable of classes in the package if the instance variable also has the default protection. As we discussed in Chapter 3, the ability to reference only those items to which a class has access is a key part of the security restrictions Java places on a program to ensure memory and API integrity.

Let's assume that no class loader based package separation exists and that we rely on Sun Microsystems to name its classes `com.sun.Car` and so on. Everything would proceed reasonably, until we surf to *www.EvilSite.org*, where someone has placed a class called `com.sun.DoSomethingEvil`. Without the namespace separation introduced by the class loader, this class would suddenly have access to all the default and protected classes (and to the default and protected variables) of every class that had been downloaded from Sun. Worse, that site could supply a class called `com.sun.Car` with a much different implementation than Sun's −− such that when the user (metaphorically, of course) applied the car's brakes, the new implementation sped up instead. Clearly, this is not a desirable situation.

Note too that with a badly written class loader, the hackers at *EvilSite.org* have the potential to supply new classes to override the core classes of the Java API. When the class loader that loaded the applet from *EvilSite* is asked to provide the `java.lang.String` class, it must provide the expected version of that class and not some version from *EvilSite.org*. In practice, this is not a problem because the class loader is written to find and return the core class first.

Without enforcement of the namespace separation that we've just outlined, there is no way to ensure that the hackers at *EvilSite.org* have not forged a class into the `com.sun` package. The only way to prevent such forgeries would be to require that every class be a signed class which authenticated that it did in fact come from `sun.com` (or wherever its package name indicates that it should have come from). Authenticated classes certainly have their place in Java's security model, but it would be unmanageable to require that every site sign and authenticate every class on its site.

Hence, the separation of classes based on the class loader that loaded them −− and the convention that applets on different pages are loaded by different class loaders −− has its benefits for Java security as well as solving a messy logistical problem. We'll now look into the details of how the class loader actually works.

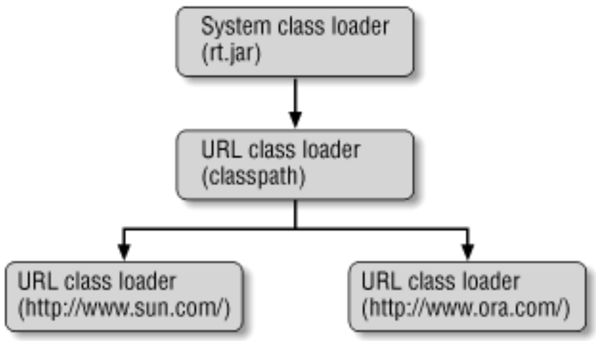
6.2 Class Loading Architecture

We'll show some examples of how to create and use class loaders a bit later in this chapter. First, let's examine from a logical perspective how class loaders work.

Class loaders are organized into a tree hierarchy. At the root of this tree is the system class loader. This class loader is also called the primordial class loader or the null class loader. It is used only to load classes from the core Java API.

The system class loader has one or more children. It has at least one child; the URL class loader that is used to load classes from the classpath. It may have other direct children, though typically any other class loaders are children of the URL class loader that reads the classpath. An example class loader hierarchy is shown in Figure 6−2; this is the hierarchy that might exist within the Java Plug−in after it has loaded applets from both *www.sun.com* and *www.ora.com*.

Figure 6−2. A class loader hierarchy



The hierarchy comes into play when it is time to load a class. Classes are loaded in one of three ways: either explicitly by calling the `loadClass( )` method of a class loader, explicitly by calling the `Class.forName( )` method, or implicitly when they are referenced by an already−loaded class.

In any case, a class loader is asked to load the class. In the first case, the class loader is the object on which the `loadClass( )` method is invoked. In the case of the `forName( )` method, the class loader is either passed to that method, or it is the class loader that loaded the class that is calling the `forName( )` method. The implicit case is similar: the class loader that was used to load the referencing class is also used to load the referenced class.

Class loaders are responsible for asking their parent to load a class; only if that operation fails will the class loader attempt to define the class itself. Take the case of the class `com.sun.Car` that was loaded by the URL class loader that knows about *www.sun.com*. When that `Car` class references the `java.lang.String` class, the same URL class loader will be asked to provide the `String` class object. It will ask its parent (the URL class loader for the classpath) to provide that class. Its parent will in turn ask the system class loader to provide that class. Since the system class loader knows about the `java.lang.String` class, it will return the appropriate class.

Note, however, that if the `com.sun.Car` class references the `com.ora.Ferrari` class, class loading will fail: the class loader that is associated with *www.ora.com* is not in the ancestor path of the class loader that is associated with *www.sun.com*.

The net effect of this is that system classes will always be loaded from the system class loader, classes on the class path will always be loaded by the class loader that knows about the classpath, and in general, a class will be loaded by the oldest class loader in the ancestor hierarchy that knows where to find a class.

When you create a class loader, you can insert it anywhere into the hierarchy of class loaders (except at the root). Typically, when a class loader is created, its parent is the class loader of the class that is instantiating the new class loader.

6.3 Implementing a Class Loader

Now we'll look at how to implement a class loader. The class loader we implement will be able to extend the normal permissions that are granted via policy files, and it will enforce certain optional security features of the class loader.

6.3.1 Class Loader Classes

The basic class that defines a class loader is the `ClassLoader` class (`java.lang.ClassLoader`):

public abstract class ClassLoader

Turn a series of Java bytecodes into a class definition. This class does not define how the bytecodes are obtained but provides all other functionality needed to create the class definition.

However, the preferred class to use as the basis of a class loader is the `SecureClassLoader` class (`java.security.SecureClassLoader`):

public class SecureClassLoader extends ClassLoader

Turn a series of Java bytecodes into a class definition. This class adds secure functionality to the `ClassLoader` class, but it still does not define how bytecodes are obtained. Although this class is not abstract, you must subclass it in order to use it.

The secure class loader provides additional functionality in dealing with code sources and protection domains. You should always use this class as the basis of any class loader you work with; in fact, the `ClassLoader` class would be private were it not for historical reasons.

There is a third class in this category: the `URLClassLoader` class (`java.net.URLClassLoader`):

public class URLClassLoader extends SecureClassLoader

Load classes securely by obtaining the bytecodes from a set of given URLs.

If you're loading classes through the filesystem or from an HTTP server, then the `URLClassLoader` provides a complete definition of a class loader. In addition, you can override some of its methods if you want to modify the security policy of classes that it defines.

6.3.2 Key Methods of the Class Loader

The `ClassLoader` class and its subclasses have three key methods that you work with when creating your own class loader.

6.3.2.1 The loadClass( ) method

The `loadClass( )` method is the only public entry into the class loader:

public Class loadClass(String name)

Load the named class. A `ClassNotFoundException` is thrown if the class cannot be found.

This is the simplest way to use a class loader directly: it requires that the class loader be instantiated and then be used via the `loadClass( )` method. Once the `Class` object has been constructed, there are three ways in which a method in the class can be executed:

- A static method of the class can be executed. This is the technique the Java virtual machine uses to execute the `main( )` method of a Java application once the initial class has been loaded, but this is not generally a technique used by Java applications.
- An object of the class can be constructed using the `newInstance( )` method of the `Class` class, but only if the class has an accessible constructor that requires no arguments. Once the object has been constructed, methods with well−known signatures can be executed on the object. This is the technique that a program like `appletviewer` uses: it loads the initial class of the applet, constructs an instance of the applet (which calls the applet's no−argument constructor), and then calls the applet's `init( )` method (among other methods). It is also used by factory classes throughout Java.
- The reflection API can be used to call a static method on the class or to construct instances of the object and execute methods on that object. The reflection API allows more flexibility than the second choice since it allows arguments to be passed to the constructor of the object.

In addition, every time that a class needs the definition of any other class, it calls the `loadClass( )` method of its class loader.

The correct implementation of the `loadClass( )` method is crucial to the security of the virtual machine. For instance, one operation this method performs is to call the parent class loader to see if it has already defined a particular class; this allows all the core Java classes to be loaded by the primordial class loader. If that operation is not performed correctly, security could suffer. As a developer you should be careful when you override this method; as an administrator, this is one of the reasons to prevent untrusted code from creating a class loader.

6.3.2.2 The findClass( ) method

The `loadClass( )` method performs a lot of setup and bookkeeping related to defining a class, but from a developer perspective, the bulk of the work in creating a `Class` class object is performed by the `findClass( )` method:

protected Class findClass(String name)

Load the class specified in the name parameter. The name will be the fully−qualified package name of the class (e.g., `java.lang.String`).

The `findClass( )` method uses whatever mechanism it deems appropriate to load the class (e.g., by reading a class file from the file system or from an HTTP server). It is then responsible for creating the protection domain associated with the class and using the next method to create the `Class` class object.

6.3.2.3 The defineClass( ) methods

These methods all take an array of Java bytecodes and some information that specifies the permissions associated with the class represented by those bytecodes. They all return the `Class` class object:

protected final Class defineClass(String name, byte[] b, int off, int len) throws ClassFormatError
protected final Class defineClass(String name, byte[] b, int off, int len, ProtectionDomain protectionDomain) throws ClassFormatError
protected final Class defineClass(String name, byte[] b, int off, int len, CodeSource cs) throws ClassFormatError

Create a class based on the bytecodes in the given array. The protection domain associated with the class varies based on which form is used:

◊ In the first method, the class is assigned to the default protection domain.
◊ In the second method, the class is assigned to the given protection domain.
◊ In the third method, the protection domain is defined by the class loader based on the given code source.

The third method in this list is available only within the `SecureClassLoader` class and its subclasses (including the `URLClassLoader` class). When you use that method, the class loader will ask what permissions are associated with a particular code source by calling this method:

protected PermissionCollection getPermissions(CodeSource cs)

Return the permissions that should be associated with the given code source. The default implementation of this method calls the `getPermissions( )` method of the `Policy` class.

Note that this gives you two effective ways in which to override the policy set up by policy files: by supplying your own `Policy` class or by supplying your own secure class loader that overrides this method.

6.3.3 Responsibilities of the Class Loader

When you implement a class loader, you override some or all of the methods we've just listed. In sum, the class loader must perform the following steps:

1. The security manager is consulted to see if this program is allowed to access the class in question. If it is not, a security exception is thrown. This step is optional; it should be implemented at the beginning of the `loadClass( )` method. This corresponds to the use of the `accessClassInPackage` permission.
2. If the class loader has already loaded this class, it finds the previously defined class object and returns that object. This step is built into the `loadClass( )` method.
3. Otherwise, the class loader consults its parent to see if the parent knows how to load the class. This is a recursive operation, so the system class loader will always be asked first to load a class. This prevents programs from providing alternate definitions of classes in the core API (but a clever class loader can defeat that protection). This step is built into the `loadClass( )` method.
4. The security manager is consulted to see if this program is allowed to create the class in question. If it is not, a security exception is thrown. This step is optional; if implemented, it should appear at the

beginning of the `findClass( )` method. Note that this step should take place after the parent class loader is queried rather than at the beginning of the operation (as is done with the access check). No Sun−supplied class loader implements this step; it corresponds to the `defineClassInPackage` permission.

5. The class file is read into an array of bytes. The mechanism by which the class loader reads the file and creates the byte array will vary depending on the class loader (which, after all, is one of the points of having different class loaders). This occurs in the `findClass( )` method.
6. The appropriate protection domain is created for the class. This can come from the default security model (i.e., from the policy files), and it can be augmented (or even replaced) by the class loader. Alternately, you can create a code source object and defer definition of the protection domain. This occurs in the `findClass( )` method.
7. Within the `findClass( )` method, a `Class` object is constructed from the bytecodes by calling the `defineClass( )` method. If you used a code source in step 6, the `getPermissions( )` method will be called to find the permissions associated with the code source. The `defineClass( )` method also ensures that the bytecodes are run through the bytecode verifier.
8. Before the class can be used, it must be resolved −− which is to say that any classes that it immediately references must also be found by this class loader. The set of classes that are immediately referenced contains any classes that the class extends as well as any classes used by the static initializers of the class. Note that classes that are used only as instance variables, method parameters, or local variables are not normally loaded in this phase: they are loaded when the class actually references them (although certain compiler optimizations may require that these classes be loaded when the class is resolved). This step happens in the `loadClass( )` method.

In the next two sections, we'll see how this plays out in each of the class loader types. Note that we do not show how to subclass the `ClassLoader` class directly: all class loaders should subclass the `SecureClassLoader` class or its subclasses instead.

6.3.4 Using the URL Class Loader

If you want to use a custom class loader, the easiest route is to use the URL class loader. This limits the number of methods that you have to override.

To construct an instance of this class, use one of the following constructors:

public URLClassLoader(URL urls[])
public URLClassLoader(URL urls[], ClassLoader parent)

Construct a class loader based on the given array of URLs. This class loader attempts to find a class by searching each URL in the order in which it appears in the array.

The parent of this class loader will be the class loader passed to the constructor or, if one is not provided, the class loader of the class that is creating the `URLClassLoader` object.

An instance of the `URLClassLoader` class may also be obtained via one of these methods:

public static URLClassLoader newInstance(URL[] urls)
public static URLClassLoader newInstance(URL[] urls, ClassLoader parent)

Create and return a URL class loader. The difference between these methods and constructing a URL class loader directly is that the class loader returned from these methods will call the security manager's `checkPackageAccess( )` method before it attempts to define a class. Only class loaders obtained this way will perform that optional step (unless you write your own class loader to perform that step).

So a URL class loader that you construct directly will not implement step 1 in the list above, while one obtained from the `newInstance( )` method will. Neither implementation provides step 4 (calling the `checkPackageDefinition( )` method of the security manager).

We can construct a URL class loader like this:

```
URL urls[] = new URL[2];
urls[0] = new URL("http://piccolo.East/~sdo/");
urls[1] = new URL("file:/home/classes/LocalClasses.jar");
ClassLoader parent = this.getClass().getClassLoader(  );
URLClassLoader ucl = new URLClassLoader(urls, parent);
```

When we use this class loader to load the class `com.sdo.Car`, the class loader first attempts to load it via *http://piccolo.East/~sdo/com/sdo/Car.class*; if that fails, it looks for the class in the *LocalClasses.jar* file.

This class loader is the basis of the class loader used by the command−line interpreter. In that case, the array of URLs is created based on the list of URLs that make up the classpath.

To implement a URL class loader, we follow the steps listed before.

6.3.4.1 Step 1: Optionally call the checkPackageAccess( ) method

If you need to modify other behavior of the URL class loader, then you cannot use the `newInstance( )` method. In that case, in order to use the `checkPackageAccess( )` method, you must override the `loadClass( )` method like this:

```
public final synchronized Class loadClass(String name, boolean resolve)
                             throws ClassNotFoundException {
    // First check if we have permission to access the package.
    SecurityManager sm = System.getSecurityManager(  );
    if (sm != null) {
        int i = name.lastIndexOf('.');
        if (i != −1) {
            sm.checkPackageAccess(name.substring(0, i));
        }
    }
    return super.loadClass(name, resolve);
}
```

6.3.4.2 Step 2: Use the previously−defined class, if available

The `loadClass( )` method of the `ClassLoader` class performs this operation for you, which is why we've called the `super.loadClass( )` method.

6.3.4.3 Step 3: Defer class loading to the parent

The `loadClass( )` method of the `ClassLoader` class performs this operation.

6.3.4.4 Step 4: Optionally call the checkPackageDefinition( ) method

In order to call the `checkPackageDefinition( )` method, you must override the `findClass( )` method:

```
protected Class findClass(final String name)
                throws ClassNotFoundException {
    // First check if we have permission to access the package.
    SecurityManager sm = System.getSecurityManager(  );
    if (sm != null) {
        int i = name.lastIndexOf('.');
        if (i != −1) {
            sm.checkPackageDefinition(name.substring(0, i));
        }
    }
    return super.findClass(name);
}
```

6.3.4.5 Step 5: Read in the class bytes

The URL class loader performs this operation for you by consulting the URLs that were passed to its constructor. If you need to adjust the way in which the class bytes are read, you should use the `SecureClassLoader` class instead.

6.3.4.6 Step 6: Create the appropriate protection domain

The URL class loader will create a code source for each class based on the URL from which the class was loaded and the signers (if any) of the class. The permissions associated with this code source will be obtained by using the `getPermissions( )` method of the `Policy` class, which by default will return the permissions read in from the active policy files. In addition, the URL class loader will add additional permissions to that set:

- If the URL has a file protocol, it must specify a file permission that allows all files that descend from the URL path to be read. For example, if the URL is *file:///xyz/classes/*, then a file permission with a name of `/xyz/classes/−` and an action list of `read` will be added to the set of permissions. If the URL is a jar file (*file:///xyz/MyApp.jar*), the name file permission will be the URL itself.
- If the URL has an HTTP protocol, then a socket permission to connect to or accept from the host will be added. For example, if the URL is *http://piccolo/classes/*, then a socket permission with a name of `piccolo:1−` and an action list of `connect,accept` will be added.

If you want to associate different permissions with the class, then you should override the `getPermissions( )` method. For example, if we wanted the above rules to apply and also allow the class to exit the virtual machine, we'd use this code:

```
protected PermissionCollection getPermissions(CodeSource codesource) {
    PermissionCollection pc = super.getPermissions(codesource);
    pc.add(new RuntimePermission("exitVM"));
    return pc;
}
```

We could completely change the permissions associated with the class (bypassing the `Policy` class altogether) by constructing a new permission collection in this method rather than calling `super.getPermissions( )`. The URL class loader will use whatever permissions are returned from this `getPermissions( )` method to define the protection domain that will be associated with the class.

6.3.4.7 Steps 7−8: Define the class, verify it, and resolve it

These steps are handled internally by the `URL` class loader.

6.3.5 Using the SecureClassLoader Class

If you need to load bytes from a source that is not a URL (or from a URL for which you don't have a protocol handler, like FTP), then you'll need to extend the `SecureClassLoader` class. A subclass is required because the constructors of this class are protected, and in any case you need to override the `findClass( )`

method in order to load the class bytes.

The subclass can call either of these constructors:

protected SecureClassLoader( )
protected SecureClassLoader(ClassLoader parent)

Create a secure class loader. The parent argument will become the parent of this class loader; if none is provided, then the class loader that calls the constructor will be the parent class loader.

The steps to use this class are exactly like the steps for the `URLClassLoader` class, except for step 5. To implement step 5, you must override the `findClass( )` method like this:

```
protected Class findClass(final String name) throws ClassNotFoundException {
    // First check if we have permission to access the package.
    // You could remove these 7 lines to skip the optional step 4.
    SecurityManager sm = System.getSecurityManager(  );
    if (sm != null) {
        int i = name.lastIndexOf('.');
        if (i != -1) {
            sm.checkPackageDefinition(name.substring(0, i));
        }
    }
    // Now read in the bytes and define the class
    try {
        return (Class)
            AccessController.doPrivileged(
                new PrivilegedExceptionAction(  ) {
                    public Object run(  ) throws ClassNotFoundException {
                        byte[] buf = null;
                        try {
                            // Acutally load the class bytes
                            buf = readClassBytes(name);
                        } catch (Exception e) {
                            throw new ClassNotFoundException(name, e);
                        }
                        // Create an appropriate code source
                        CodeSource cs = getCodeSource(name);
                        // Define the class
                        return defineClass(name, buf,
                                           0, buf.length, cs);
                    }
                }
            );
    } catch (java.security.PrivilegedActionException pae) {
        throw (ClassNotFoundException) pae.getException(  );
    }
}
```

The syntax of this method is complicated by the fact that we need to load the class bytes in a privileged block. Depending on your circumstances, that isn't strictly necessary, but it's by far the most common case for class loaders. Say that your class loader loads class `A` from the database; that class is given minimal permissions. When that class references class `B`, the class loader will be asked to load class `B` and class `A` will be on the stack. When it's time to load the new class bytes, we need to load them with the permissions of the class loader rather than the entire stack, which is why we use a privileged block.

Notwithstanding, the `try` block has three operations: it loads the class bytes, it defines a code source for that class, and it calls the `defineClass( )` method to create the class. The first two of the operations are encapsulated in the `readClassBytes( )` and `getCodeSource( )` methods; these are methods that you must implement.

Loading the class bytes is an operation left to the reader. The reason for providing your own class loader is that you want to read the class bytes in some special way; otherwise, you'd use the `URLClassLoader` class. The code source is another matter: we must determine a URL and a set of certificates that should be associated with the class.

In a signed jar file, the certificates are read from the jar file and the URL is the location of the jar file. In Chapter 12, we'll show how to get the certificates from a standard jar file and construct the appropriate code source. If your class definition isn't coming from a URL, then you must be a little creative. The simplest approach is to create an arbitrary URL. You could also use the methods we examine in Chapter 10, and load one or more certificates from the keystore; you would then use both items to construct the code source.

The `defineClass( )` method will call back to the `getPermissions( )` method in order to complete the definition of the protection domain for this class. And that's why the URL used to construct the code source can be arbitrary: when you write the `getPermissions( )` method, just make sure that you understand what the URL actually is. In default usage, the URL would be used to find entries in the policy files, but since you're defining your own permissions anyway, the contents of the URL don't matter. What matters is that you follow a consistent convention between the definition of your `getCodeSource( )` and `findClass( )` methods.

Hence, possible implementations of the `getPermissions( )` and `getCodeSource( )` methods are as follows:

```
protected CodeSource getCodeSource(String name) {
    try {
        return new CodeSource(new URL("file", "localhost", name),
                              null);
    } catch (MalformedURLException mue) {
        mue.printStackTrace(  );
    }
    return null;
}

protected PermissionCollection getPermissions(CodeSource codesource) {
    PermissionCollection pc = new Permissions(  );
    pc.add(new RuntimePermission("exitVM"));
    return pc;
}
```

If you're reading the class bytes from, say, a database, it would be more useful if you could pass an arbitrary string to construct the code source. That doesn't work directly since the code source requires a URL but the file part of the URL can be any arbitrary string. In this case, we just use the class name.

Note that the `getPermissions( )` method of the `SecureClassLoader` class does not add the additional permissions that the same method of the `URLClassLoader` class adds. As a result, we do not call the `super.getPermissions( )` method; instead, we construct a new permissions object directly.

6.3.6 Other Class Loaders

There are other class loaders within the Java API. Classes loaded by the primordial class loader do not have an associated protection domain (alternately, we may say that they are associated with the system protection domain), which is why they have permission to perform any operation.

The `appletviewer` and Java Plug−in define their own class loader, which is an extension of the URL class loader. This class loader extends to classes loaded from a file URL the permissions to open a connection to and accept a connection from the localhost. Classes loaded from an HTTP URL have the same permissions as those granted in the URL class loader.

The only other publicly−accessible class loader in the core API is the RMI class loader. There is a class called `RMIClassLoader` (`java.rmi.server.RMIClassLoader`) which, despite its name, is neither a class loader nor restricted to RMI. This class has a static method called `loadClass( )` which, like the `loadClass( )` method of the `ClassLoader` class, finds the named class and defines it. It uses an internal class loader to do this; the internal class loader happens to be a modification of the `URLClassLoader` class. The URL used by this class loader is specified by the `java.rmi.server.codebase` property; it uses the same permissions as a standard URL class loader. If the features of this class loader meet your requirements, you can use it in any program, regardless of whether your program uses RMI.

6.4 Miscellaneous Class Loading Topics

There are a few details about class loaders that we haven't yet covered. These details are not directly related to the security aspects of the class loader, which is why we've saved them until now. If you're interested in the complete details of the class loader, we'll fill in the last few topics here.

6.4.1 Delegation

As we've mentioned, class loading follows a delegation model. This model permits a class loader to be instantiated with this constructor:

protected ClassLoader(ClassLoader parent)

Create a class loader that is associated with the given class loader. This class loader delegates all operations to the parent first: if the parent is able to fulfill the operation, this class loader takes no action. For example, when the class loader is asked to load a class via the `loadClass( )` method, it first calls the `loadClass( )` method of the parent. If that succeeds, the class returned by the delegate will ultimately be returned by this class. If that fails, the class loader then uses its original logic to complete its task, something like this:

```
public Class loadClass(String name) {
        Class cl;
        cl = delegate.loadClass(name);
        if (cl != null)
                return cl;
        // else continue with the loadClass(  ) logic
}
```

You may retrieve the delegate associated with a class loader with the following method.

public final ClassLoader getParent( )

Return the class loader to which operations are being delegated.

The class loader that exists at the root of the class loader hierarchy is retrieved via this method:

public static ClassLoader getSystemClassLoader( )

Return the system class loader (the class loader that was used to load the base application classes). If a security manager is in place, you must have the `getClassLoader` runtime permission to use this method.

6.4.2 Loading Resources

A class loader can load not only classes, but any arbitrary resource: an audio file, an image file, or anything else. Instead of calling the `loadClass( )` method, a resource is obtained by invoking one of these methods:

public URL getResource(String name)
public InputStream getResourceAsStream(String name)
public URL findResource(String name)

Find the named resource and return either a URL reference to it or an input stream from which it can be read. Implementations of class loaders should look for resources according to their internal rules, which are typically (but need not be) the same rules as are used to find classes.

The `getResource( )` method calls the `getSystemResource( )` method; if it does not find a system resource, it returns the object retrieved by a call to the `findResource( )` method (which by default will be `null`). The `getResourceAsStream( )` method simply calls the `getResource( )` method and, if a resource is found, opens the stream associated with the URL.

public static URL getSystemResource(String name)
public static InputStream getSystemResourceAsStream(String name)

Find the named resource and return either a URL reference to it or an input stream from which it can be read. By default, these methods look for the resource on the classpath and return that resource (if found).

public final Enumeration getResources(String name)
public Enumeration findResources(String name)

Return an enumeration of resources with the given name. In the first method, an enumeration of the local resources of all delegated class loaders (including the present class loader) is returned; in the second method, only the local resources of the present class loader are returned.

6.4.3 Loading Libraries

Loading classes with native methods creates a call to this method of the `ClassLoader` class:

protected String findLibrary(String libname)

Return the directory from which native libraries should be loaded.

This method is used by the `System.loadLibrary( )` method to determine the directory in which the native library in question should be found. If this method returns `null` (the default), the native library must be in one of the directories specified by either the `java.library.path` or `java.sys.library.path` property; typically, these properties are set in a platform−specific way (e.g., from the `LD_LIBRARY_PATH` on Solaris or the `PATH` on Microsoft Windows).

However, custom class loaders can override that policy and require that libraries be found in some application−defined location. This prevents a user from overriding the runtime environment to specify an alternate location for that library, which offers a slight security advantage. Note that if the user can write to the hardwired directory where the library lives, this advantage no longer exists: the user can simply overwrite the existing library instead of changing an environment variable to point to another library; the end result is the same.

6.5 Comparison with Previous Releases

In Java 2, version 1.3, no class loader can define a class that is in the java package (i.e., whose class name begins with `java`). In 1.2 and earlier releases, that restriction is not present (unless you code it into your own class loader). That's one reason why there is no class loader that uses the `checkPackageDefinition( )` method (although that method was not used in 1.2 either).

In Java 1.1, class loading was significantly different. To begin, there are no code source or protection domain objects in 1.1, and classes do not carry associated permissions with them. Security in 1.1 is solely up to the security manager, and the class loader simply loads classes.

That means that the `loadClass( )` method in 1.1 is very different. In particular, there is only one `defineClass( )` method that it can call (the one without a protection domain argument), and it does not call the `findClass( )` method. In order to write a class loader in 1.1, you must override the `loadClass( )` method and perform all your work in that method. We show an example of this in Appendix D.

The secure class loader and URL class loader were introduced in Java 2, version 1.2. In 1.1, the primordial class loader is used to load all classes on the classpath in addition to the classes in the core API.

Delegation was also introduced in Java 2, version 1.2. In 1.1, a custom class loader can make a special method call to load classes from the classpath, but there is no general hierarchy of class loaders.

In 1.1, the default behavior of the methods that retrieve resources is to return `null`.

The RMI class loader does not exist in Java 1.0.

6.6 Summary

The class loading mechanism is integral to Java's security features. Typically this integration is considered in light of the relationship between the class loader, the access controller, and the security manager. However, the class loader is important in its own right. The class loader must enforce the namespace separation between classes that are loaded from different sites (especially when these different sites are untrusted); this helps to enforce the security mechanisms of the Java language.

For sites that need a more flexible security policy, a custom class loader may be desirable. Custom class loaders allow the security policy to be modified as classes are defined; this is similar to (and compatible with) providing a new implementation of the `Policy` class. However, custom class loaders can bypass the policy class altogether, which means that they can define immutable security policies (though, of course, installing the class loader in the first place still requires that applications have the appropriate policy−based permissions). In certain circumstances, this is easier than modifying and installing a new `Policy` class.

Chapter 7. Introduction to Cryptography

So far, we've examined the basic level of Java's security paradigm −− essentially, those features that make up the Java sandbox. We're now going to shift gears somewhat and begin our examination of the cryptographic features in the Java security package itself. The Java security package is a set of classes that were added to Java 1.1 (and expanded in Java 2, version 1.2); these classes provide additional layers of security beyond the layers we've examined so far. Although these classes do play a role in the Java sandbox −− they are the basis on which Java classes may be signed, and expanding the sandbox based on signed classes is a key goal of Java security −− they may play other roles in secure applications.

A digital signature, for example, can authenticate a Java class so that the security policy can allow that class greater latitude in the operations it can perform, but a digital signature is a useful thing in its own right. An HR department may want to use a digital signature to verify requests to change payroll data, an online subscription service might require a digital signature to process a change order, and so on. Thus, while we'll examine the classes of the Java security package from the perspective of what we'll be able to do with a signed class, the techniques we'll show will have broader applicability.

In order to use the classes of the security package, you don't need a deep understanding of cryptographic theory. This chapter will explain the basic concepts of the operations involved, which should be sufficient to understand how to use the APIs involved. On the other hand, one feature of the security package is that different implementations of different algorithms may be provided by third−party vendors. We'll explain how to go about providing such implementations, but it is assumed that readers who are interested in writing such an implementation already understand the mechanics of cryptography. Hence, we won't give any cryptographically valid examples in those sections. If you're interested in this type of information, a good reference is Jonathan Knudsen's *Java Cryptography* (O'Reilly & Associates).

If you already have an understanding of the basics of digital signatures, encryption, and the need for authentication, you can skip this chapter, which provides mainly background information.

7.1 The Need for Authentication

We are primarily concerned with one goal of the security package: the ability to authenticate classes that have been loaded from the network. The components of the Java API that provide authentication may have other uses in other contexts (including within your own Java applications), but their primary goal is to allow a Java application (and the Java Plug−in) to load a class from the network and be assured of two things:

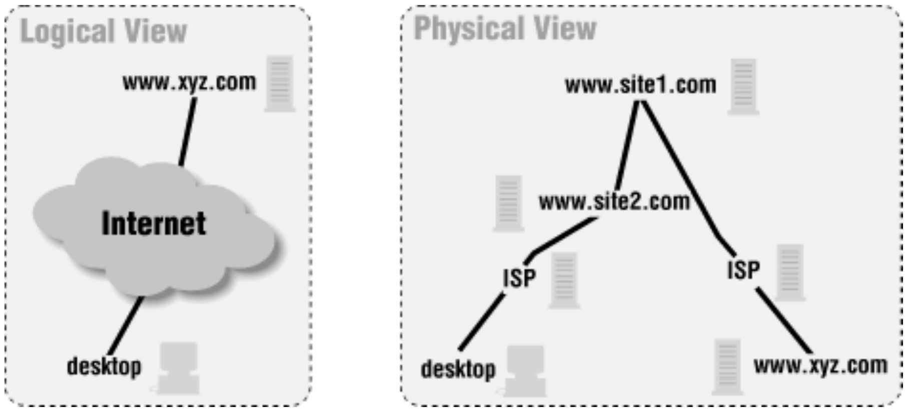
- The identity of the site from which the class was loaded can be verified (author authentication).
- The class was not modified in transit over the network (data authentication).

As we've seen, Java applications typically assume that all classes loaded over the network are untrusted classes, and these untrusted classes are generally given permissions consistent with that assumption. Classes that meet the above two criteria, however, need not necessarily be so constrained. If you walk into your local software store and buy a shrink−wrapped piece of software, you're generally confident that the software will not contain viruses or anything else that's harmful. This is part of the implied contract between a commercial software producer and a commercial software buyer. If you download code from that same software producer's web site, you're probably just as confident that the code you're downloading is not harmful; perhaps it should be given the same access rights as the software you obtained from that company through a more traditional channel.

There's a small irony here because many computer viruses are spread through commercial software. That's one reason why the fact that a class has been authenticated does not necessarily mean it should be able to access anything on your machine that it wants to. It's also a reason why the fine−grained nature of the access controller is important: if you buy classes from *acme.com* but only give them access to certain things on your machine, you are still somewhat protected if by mistake *acme.com* includes a virus in their software.

Even if all commercial software were virus−free, however, there is a problem with assuming that code downloaded from a commercial site is safe to run on your machine. The problem with that assumption −− and the reason that Java by default does not allow that assumption to be made −− has to do with the way in which the code you execute makes its way through the Internet. If you load some code from *www.xyz.com* onto your machine, that code will pass through many machines that are responsible for routing the code between your site and XYZ's site. Typically, we like to think that the data that passes between our desktop and *www.xyz.com* enters some large network cloud; it's called a cloud because it contains a lot of details, and the details aren't usually important to us. In this case, however, the details are important. We're very interested to know that the data between our desktop and *xyz.com* passes through, for example, our Internet service provider, two other sites on the Internet backbone, and XYZ's Internet service provider. Such a transmission is shown in Figure 7−1. The two types of authentication that we mentioned above provide the necessary assurance that the data passing through all these sites is not compromised.

Figure 7−1. How data travels through a network



7.1.1 Author Authentication

First we must prove that the author of the data is who we expect it to be. When you send data that is destined for *www.xyz.com*, that data is forwarded to *site2*, who is supposed to forward it to *site1*, who should simply forward it to XYZ's Internet service provider. You trust *site1* to forward the data to XYZ's Internet service provider unchanged; however, there's nothing that causes *site1* to fulfill its part of this contract. A hacker at *site1* could arrange for all the data destined for *www.xyz.com* to be sent to the hacker's own machine, and the hacker could send back data through *site2* that looked as if it originated from *www.xyz.com*. The hacker is now successfully impersonating the *www.xyz.com* site. Hence, although the URL in your browser says *www.xyz.com*, you've been fooled: you're actually receiving whatever data the impersonator of XYZ Corporation wants to send to you.

There are a number of ways to achieve this masquerade, the most well−known of which is DNS spoofing. When you want to surf to *www.xyz.com*, your desktop asks your DNS server (which is typically your Internet service provider) for the IP address of *www.xyz.com* and you then send off the request to whatever address you receive. If your Internet service provider knows the IP address of *www.xyz.com*, it tells your desktop what the correct address is; otherwise, it has to ask another DNS server (e.g., *site1*) for the correct IP address. If a hacker has control of a machine anywhere along the chain of DNS servers, it is relatively simple for that hacker to send out his own address in response to a DNS request for *www.xyz.com*.

Now say that you surf to *www.xyz.com* and request a Java class (or set of classes) to run a spellchecker for your Java−based word processor. The request you send to *www.xyz.com* will be misaddressed by your machine −− your machine will erroneously send the request to the hacker's machine since that's the IP address your machine has associated with *www.xyz.com*. Now the hacker is able to send you back a Java class. If that Java class is suddenly trusted (because, after all, it allegedly came from a commercial site), it has access that you wouldn't necessarily approve: perhaps while it's spellchecking your document, it is also searching your hard disk to find the datafile of your financial planning software so that it can read that file and send its contents back over the network to the hacker's machine.

Yes, we've made this sound easier than it is −− the hacker would have to have intimate knowledge of the *xyz.com* site to send you back the classes you requested, and those classes would have to have the expected interface in order for any of their code to be executed. But such situations are not difficult to set up either; if the hackers stole the original class files from *www.xyz.com* −− which is usually extremely easy −− all they need to do is set themselves up at the right place in the DNS chain.

In the strict Java security model we explored earlier, this sort of situation is possible, but it is not dangerous. Because the classes loaded from the network are never trusted at all, the class that was substituted by the hackers is not able to damage anything on your machine. At worst, the substituted class does not behave as you expect and may in fact do something quite annoying −− like play loud music on your machine instead of spellchecking your document. But the class is not able to do anything dangerous, simply because all classes from the network are untrusted.

In order to trust a class that is loaded from the network, then, we must have some way to verify that the class actually came from the site it said it came from. This authentication comes from a digital signature that comes with the class data −− an electronic verification that the class did indeed come from *www.xyz.com*.

7.1.2 Data Authentication

The second problem introduced by the fact that our transmissions to *www.xyz.com* must pass through several hosts is the possibility of snooping. In this scenario, assume that *site2* on the network is under control of a hacker. When you send data to *www.xyz.com*, the data passes through the machine on *site2*, where the hacker can modify it; when data is sent back to you, it travels the same path, which means that the hacker on *site2* can again modify the data.

This lack of privacy in data transmission is one reason you might want data over the network to be encrypted −− certainly if the spellchecking software you're using from *www.xyz.com* is something you must pay for, you don't want to send your unencrypted credit card number through the network so that *site2* can read it. However, for authentication purposes, encrypting the data is not strictly necessary. All that is necessary is some sort of assurance that the data that has passed through the network has not been modified in transit. This can be achieved by various cryptographic algorithms even though the data itself is not encrypted. The simpler path is to use such a cryptographic algorithm (known as a message digest algorithm or a digital fingerprint) instead of encrypting the data.

Encryption Versus Data Authentication

When you send data through a public network, you can use a digital fingerprint of that data to ensure that the data was not modified while it was in transit over the network. This fingerprint is sufficient to prevent a snooper from substituting new data (e.g., a new Java class file) for the original data in your transmission.

However, this authentication does not prevent a snooper from reading the data in your transmission; authenticated data is not encrypted data. If you are worried about someone stealing your data, the security provided by data authentication is insufficient. Data authentication prevents writing of data but not reading of data.

This can be a very important difference in countries that place import or export controls on encryption. Those restrictions do not apply to digital signatures, so the Java code that implements digital signatures is freely available. Hence, it is easier to deploy an application that requires digital signatures than one that requires encryption.

Without some cryptographic mechanism in place, the hacker at *site2* has the option of modifying the classes that are sent from *www.xyz.com*. When the classes are read by the machine at *site2*, the hacker could modify them in memory before they are sent back onto the network to be read by *site1* (and ultimately to be read by your machine). Hence, the classes that are sent need to have a digital fingerprint associated with them. As it turns out, the digital fingerprint is required to sign the class as well.

7.1.3 Java's Role in Authentication

When Java was first released and touted as being "secure," it surprised many people to discover that the types of attacks we've just discussed were still possible. As we've said, security means many things to many people, but a reasonable argument could be made that the scenarios we've just outlined should not be possible in a secure environment.

The reasons Java did not solve these problems in its first release are varied, but they essentially boil down to one practical reason and one philosophical reason.

The practical reason is that all the solutions we're about to explore depend to a high degree on technologies that are just beginning to become viable. As a practical matter, authentication relies on everyone having public keys available −− and as we'll discuss in Chapter 10, that's not necessarily the case. Without a robust mechanism to share public keys, Java had two options:

- Provide no security at all, and allow applets full use of the resources of the user's computer. By now, we know all the possible problems with that route.
- Provide the very strict security that was implemented in 1.0−based versions of Java, with a view toward ways of enhancing that model as technologies evolved. While not the best of all possible worlds, this compromise allowed Java to be adopted much sooner than it would otherwise have been.

On a philosophical level, however, there's another argument: Java shouldn't solve these problems because they are not confined to Java itself. Even if Java classes were always authenticated, that would not prevent the types of attacks we've outlined here from affecting non−Java−related transmissions. If you surf to *www.xyz.com* and that site is subject to DNS spoofing, you'll be served whatever pages the spoofer wants to substitute. If you engage in a standard non−Java, forms−based transmission with *www.xyz.com*, a snooper along the way can steal and modify the data you're sending over the standard HTTP protocol.

In other words, the attacks we've just outlined are inherent in the design of a public network, and they affect all traffic equally −− email traffic, web traffic, FTP traffic, Java traffic, and so on. In a perfect world, solving these problems at the Java level is inefficient, as it means that the same problem must still be solved for all the other traffic on the public network. Solving the problem at the network level, on the other hand, solves the problem once and for all, so that every protocol and every type of traffic are protected.

There are a number of popular technologies that solve this problem in a more general case. If all the traffic between your site and *www.xyz.com* occurs over SSL using an HTTPS−based URL, then your browser and the *www.xyz.com* web server will take care of the details of authentication of all web−based traffic, including the Java−related traffic. That solves the problem at the level of the web browser, but that still is not a complete solution. If the applet needs to open a connection back to *www.xyz.com*, it must use SSL for this communication as well. And we still have other, non−web−related traffic that is not authenticated.

It would be better still to solve this problem at the network level itself. There are many products from various vendors that allow you to authenticate (and encrypt) *all* data between your site and a remote site on the network. Using such a product is really the ideal from a design point of view; in that way, all data is protected, no matter what the source of the traffic. Either of these solutions makes authentication and fingerprinting of Java classes redundant (and may offer the benefit that the data is actually encrypted when it passes through the network).

Unfortunately, these solutions lead us back to practical considerations: if it's hard for Java environments to share digital keys and to manage cryptographic technology, it's harder still to depend on the network software to manage this process. So while it might be ideal for this problem to be solved for the network as a whole, it's impractical to expect such a solution. Hence, the Java security package offers a reasonable compromise: it allows you to deploy and use trusted (i.e., authenticated) classes, but their use is not mandated in case you prefer to employ a broader solution to this problem.

7.2 The Role of Authentication

In the preceding discussion, we assumed that you want to load classes from *www.xyz.com* and that you want those classes to be trusted so that they might have some special permission when they execute on your machine. For example, the spellchecking class might need to open up a local dictionary file to learn how to spellcheck names and other data you customized for the spell checker.

Do not, however, make the assumption that all classes that are authenticated are therefore to be trusted, or even that all trusted classes should necessarily have the same set of permissions. There's nothing that prevents me from obtaining the necessary information and tools so that I can sign and encrypt all of my classes. When you download those classes, you know with certainty that the classes came from me −− they carry my digital signature and they've been fingerprinted to ensure that they haven't been tampered with.

But that's *all* the information that you know about these classes. In particular, just because the classes were authenticated does not mean that I didn't put a virus into them that's going to erase all the files on your hard disk. And just because you know that a particular Java applet came from me does not mean that you can necessarily track me down when something goes wrong. If you surf to my home page and run my authenticated applet, then surf to *www.sun.com* and run their authenticated applet, then surf to *www.EvilSite.org* and run their authenticated applet, and then two weeks later your hard disk is erased, how will you know which site planted the delayed virus onto your machine? How will you even remember which sites you had visited in the last two weeks (or longer)? If you have an adequate set of backups and other logs, it is conceivable that you might be able to recreate what happened and know at whom to point your finger (and whom to sue), but such a task would be arduous indeed. And if the virus affected your logs, the finger of suspicion might point to the incorrect site.

Hence, the role of authentication of Java classes is not to validate that those classes are trusted or to automatically give those classes special permissions. The role of authentication is to give the user (or, for a corporate network, the system or network administrator) more information on which to base a security policy. A reasonable policy might be that classes that are known to come from *www.SpellChecker.com* can read the user's personal dictionary file −− but that doesn't mean they should necessarily be able to read anything else. A reasonable policy would also grant this type of exception to the general rule about permissions given to network classes only in very specific cases to a few well−known sites, and consider unknown but authenticated sites as still untrusted.

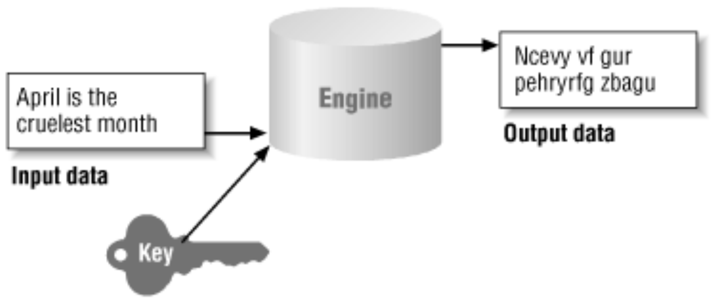
The moral of the story is that authentication does not magically solve any problem; it is merely a tool that can be used in the pursuit of solutions.

7.3 Cryptographic Engines

In the next few chapters of this book, we're going to see how Java provides an interface to the algorithms required to perform the sort of authentications we've just talked about. We'll also explore the architecture Java provides for general implementation of these algorithms, including ones (such as encryption) that are not strictly required for authentication. If you're not familiar with the various cryptographic algorithms we've been alluding to so far in this chapter, the next section should sort that all out for you.

Essentially, all cryptographic operations are structured like the diagram in Figure 7−2. Central to this idea is the cryptographic algorithm itself, which is called an engine; the term "algorithm" is reserved to refer to particular implementations of the cryptographic operation. The engine takes some set of input data and (optionally) some sort of key and produces a set of output data. A few points are relevant to this diagram. There are engines that do not require a key as part of their input. In addition, not all cryptographic engines produce symmetric output −− that is, it's not always the case that the original text can be reconstructed from the output data. Also, the size of the output is typically not the same as the size of the input. In the case of message digests and digital signatures, the output size is a small, fixed−size number of bytes; in the case of encryption engines, the output size is typically somewhat larger than the input size.

Figure 7−2. A cryptographic engine for encryption



In the Java security package, there are two standard cryptographic engines: a message digest engine and a digital signature engine. In addition, for some users, an optional engine is available to perform encryption. Finally, because keys are central to the use of most of these engines, there is a wide set of classes that operate on keys, including engines that can be used to generate certain types of keys. The term "engine" is also used within the security package to refer to other classes that support these operations.

7.3.1 Cryptographic Keys

The first engines we'll look at generate cryptographic keys. Keys are the basis for many cryptographic operations. In its simplest sense, a key is a long string of numbers −− not just any string of numbers, but a string of numbers that has very strict mathematical properties. The mathematical properties a key must have vary based on the cryptographic algorithms it is going to be used for, but there's an abstract (logical) set of properties all keys must have. It's this abstract set of properties that we'll see in the Java security package.

In the realm of cryptography, keys can either come alone (in which case they are called secret keys) or in pairs. A key pair has two keys, a public key and a private key. So all together there are three types of keys −− secret, public, and private.

When an algorithm requires a secret key, both parties using the algorithm will use the same key. Both parties must agree to keep the key secret, lest the security of the cryptography between the parties be compromised.

The secret key approach suffers from two problems. First, it requires a separate key for every pair of parties that need to send encrypted data. If you want to send your encrypted credit card data to ten different Internet stores, you would need ten different keys. Worse yet, if you operated an Internet store and had millions of customers, you would need literally millions of keys −− one per customer. Management of such keys is a very difficult problem.

The other problem with this approach is coming up with a method for sharing the keys. It's crucial that the key be kept secret, since anyone with the key can decrypt the data to be shared. Hence, you can't simply send the key over the network without somehow encrypting the key itself; doing so would be tantamount to sending the data itself unencrypted. There are engines that perform secure secret key exchange, however; these key exchanges are the basis of many operations, including SSL.

Public and private keys can provide asymmetric operation to cryptographic engines. The public key can be used by one party participating in the algorithm, and the private key can be used by the other party.

The usefulness of this type of key pair is that one key can be published to the world. You can email your public key to your friends (and your enemies), you can put it on a global key server somewhere, you can broadcast it on the Internet −− as long as you don't lose your private key, you can do anything you like with your public key.

Then, when someone wants to send you some sensitive information, they can use your public key to encrypt the data −− and, as long as you have kept your private key private, you'll be the only one who is actually able to decrypt the data. Similarly, when you want to send sensitive data to someone, all you need is their public key; when the data has been encrypted with the public key, you know that only the holder of the private key will be able to read what you've sent her. In the area of digital signatures, this key ordering is reversed: you sign a document with your private key, and the recipient of the document needs your public key in order to verify the digital signature.

Public key encryption is not without its key management problems as well, however. When you receive a digitally signed document, you need the public key of the signer of the document. The mechanism to obtain that key is very fluid; there are a number of proposals for centralized key warehouses that would hold public keys and for methods to access those keys, but the infrastructure to make this all a reality is not really in place. Hence, users of public keys have adopted a variety of techniques for obtaining the public keys.

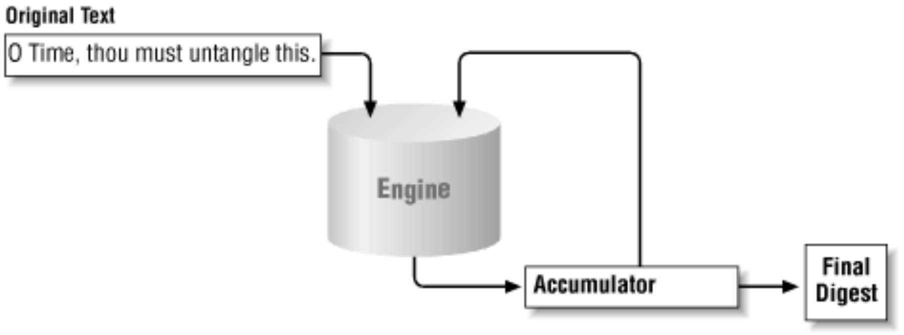
7.3.2 Message Digests

The second engines that we'll examine deal with message digests. A message digest is the digital fingerprint we alluded to earlier. Conceptually, a message digest is a small sequence of bytes that is produced when a given set of data is passed through the message digest engine. Unlike other cryptographic engines, a message digest engine does not always require a key to operate; some do, and some do not. A message digest engine takes a single stream of data as its input and produces a single output. We call the output a message digest (or simply a digest, or a hash), and we say that the digest represents the input data.

The digest that corresponds to a particular set of data does not reflect any information about that data −− in particular, there is no way to tell from a digest how much data it represents or what the data actually was. A message digest is useful only when the data it represents is also available. If you want to determine whether a particular digest represents a particular set of data, you must recalculate the digest and compare the newly calculated digest with the original digest. If the two are equal, you've verified that the original digest does indeed represent the given set of data.

Data that is fed into a message digest engine is always treated as an ordered set of bytes. If even one byte of the data is altered or absent (or presented out of order), the digest will be different. Hence, a typical message digest algorithm has an internal accumulator that operates on all data fed into the engine. As each byte of data is fed into the engine, it is combined with the data in the accumulator to produce a new value, which is stored in the accumulator to provide input (see Figure 7−3).

Figure 7−3. The message digest accumulator



As a simple example, consider a message digest algorithm based on the exclusive−or of all the input bytes. The accumulator starts with a value of 0. If the string "O Time, thou must untangle this" is passed to the engine, the engine considers the bytes one at a time.[1] The first byte, "O", has a value of 0x4f, which will *xor* with the accumulator to provide a value of 0x4f. The next byte, a space (0x20), will *xor* with the accumulator to produce a value of 0x6f. And so on, such that the final result of the accumulator is 0x67.

[1] Don't be confused by the fact that we're dealing in bytes here when the characters in a Java string are two bytes long. The data passed to the message digest engine is treated as arbitrary binary data −− it doesn't matter if the data was originally ASCII (that is, byte−oriented) data, a Java character string, or a binary class file.

There are a few differences between this example and a real message digest algorithm. First, standard algorithms typically operate on 4− or 8−byte quantities, so the bytes that are fed into the engine are first grouped into `int`s or `long`s, with padding added if the input data is not a multiple of the desired quantity. Second, they produce a digest that is usually 64 or 128 bits long rather than a single byte; this final digest may be the value left in the accumulator or it may be the value left in the accumulator subjected to additional operations.

The difference in the output size is one of the crucial differences. At best, the example we just walked through could produce 256 different digests. Any two given inputs have a 1 in 256 chance of producing the same digest, which is clearly not a sufficient guarantee that a digest represents a given set of data. In the example above, the string "O Time, thou must untangle this" produced a digest of 0x67 −− but so does the string "g". An algorithm that produces a 64−bit digest, on the other hand, produces over 18 quintillion unique digests, so the odds that two data sequences will produce the same digest are very remote indeed.

This brings us to another of the crucial differences −− a successful message digest algorithm must provide an assurance that it is computationally infeasible to find two messages that produce the same digest. This ensures that a new set of data cannot be substituted for the original data so that each produces the same digest.

Note also that a message digest in itself is not a secure entity. A digest is often provided with the data it represents; the recipient of the data then recalculates the digest to make sure that the data was not originally tampered with. But nothing in this scenario prevents someone from modifying both the original data and the digest since both are transmitted and since the calculation of the digest is a well−known operation requiring no key. However, secure message digests can be produced when you introduce a key into the mix; these types of digests are called message authentication codes.

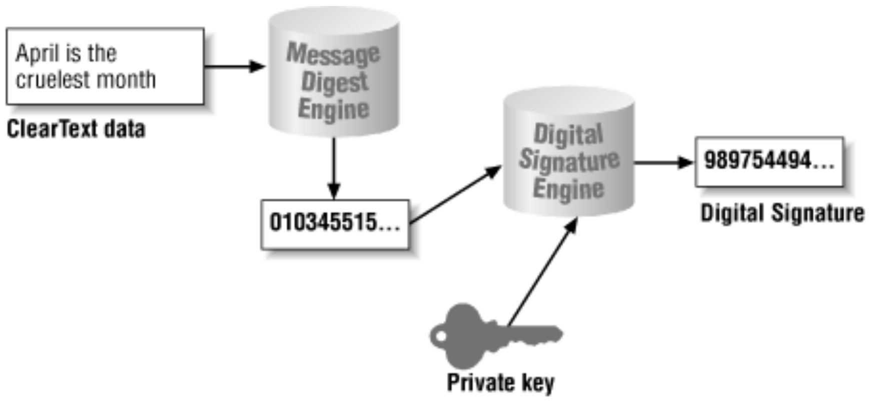
7.3.3 Digital Signatures

The primary engine in the security package (at least as far as authentication goes) is the digital signature engine. Like a real signature, a digital signature is presumed to provide a unique identification of an entity (that is, an individual or an organization). Like a real signature, a digital signature can be forged, although it's much harder to forge a digital signature than a real signature.[2] Forging a digital signature requires access to the private key of the entity whose signature is being forged; this is yet another reason why it is important to keep your private keys private. Like a real signature, a digital signature can be "smudged" so that it is no longer recognizable. And because they're based on key certificates, digital signatures have other properties, such as the fact that they can expire.

[2] On the other hand, a forged digital signature is undetectable, unlike a forged real signature.

Digital signatures rely on two things: the ability to generate a message digest and the ability to encrypt that digest. The entire process is shown in Figure 7−4.

Figure 7−4. Generating a digital signature



The process is as follows:

1. A message digest is calculated that represents the input data.
2. The digest is then encrypted with the private key.

Note that encryption is performed on the digest and not on the data itself. In order to present this signature to another entity, you must present the original data with it −− the signature is just a message digest, and, as we mentioned earlier, you cannot reconstruct the input data from the message digest.

Verifying a digital signature requires the same path; the message digest of the original data must be calculated. The signed digest is decrypted with the public key and if the decrypted digest matches the calculated digest, the signature is valid. Strictly speaking, the operations performed on the digests are not necessarily encryption and decryption; most digital signature algorithms cannot be used for encryption of arbitrary data. The symmetry of the operation is the same.

Nothing prevents the signed data from being intercepted. So the data that accompanies the digital signature cannot be sensitive data; the digital signature only verifies that the message came from a particular entity and that the message was not altered in transit, but it does not actually protect that message from being read by anyone with access to it.

If the data is altered, it will not produce the same message digest, which in turn will not produce the same digital signature. And it's computationally infeasible to change the data, generate a new digest of that data, and then regenerate the digital signature without access to the private key. It is, however, possible to replace one message that was signed by a private key with another message that was signed by that same private key.

7.3.4 Encryption Engines

The final engines we'll discuss handle actual encryption. These engines are part of the Java Cryptography Extension (JCE) and the Java Secure Socket Extension (JSSE) rather than the core security package. Encryption engines handle the encryption and decryption of arbitrary data, just as we would expect. An important thing to note is that the encryption engines that are part of JCE are not used in the generation and verification of digital signatures −− digital signatures use their own algorithms to encrypt and decrypt the message digest that are suitable only for manipulating data the size of a message digest. This difference allows the digital signature engine to be exportable, where the encryption engines may not be.

7.4 Summary

Much of the Java security package is made up of a collection of engines, the basic properties of which we've outlined in this chapter. As a unit, these engines allow us primarily to create digital signatures −− a useful notion that authenticates a particular piece of data. One thing that a digital signature can authenticate is a Java class file, which provides the basis for a security manager to consider a class to be trusted (as least to some degree), even though the class was loaded from the network.

The security package, like many Java APIs, is actually a fairly abstract interface that several implementations may be plugged into. Hence, another feature of the security package is its infrastructure to support these differing implementations. In the next chapter, we'll explore the structure of the security package and how it supports these differing implementations; we'll then proceed into how to use the engines of the security package.

Chapter 8. Security Providers

The cryptographic engines in Java that provide for digital signatures, message digests, and the like are provided as a set of abstract classes in the Java security package. Concrete implementations of these classes are provided by Sun in the SDK, and you have the option of obtaining third−party implementations of these engines. All of this is made possible through the security provider infrastructure. The provider infrastructure allows concrete implementations of various classes in the security package to be found at runtime, without any changes to the code. This provides a consistent API that can be used by all programs, regardless of who provides the actual implementation.

Java 2, version 1.3 comes with two security providers: one performs operations that implement DSA−based algorithms (plus some other default operations) and one performs operations that implement RSA−based algorithms. Sun supplies two additional security providers: one with JCE and one with JSSE. We'll discuss how to install those additional providers in this chapter and then look at the Java classes that comprise the security provider architecture.

In terms of actual programming, the classes we're going to examine in this chapter are rarely used −− hence, we will not delve much into programming. To meet the needs of most developers, end users, and administrators, this chapter focuses on the architecture of the security provider since that gives us the ability to substitute new implementations of the cryptographic engines we'll use in the rest of the book. Following that discussion, we'll move into the implementation of the architecture for those readers who are interested in the details.

8.1 The Architecture of Security Providers

The security provider abstracts two ideas: engines and algorithms. In this context, "engine" is just another word for operation; there are certain operations the security provider knows about, and in Java, these operations are known as engines. An algorithm defines how a particular operation should be executed. An algorithm can be thought of as an implementation of an engine, but that can lead to confusion because there may be several implementations of an algorithm.

As a simple example, the Java security package knows about message digests. A message digest is an engine: it is an operation a programmer can perform. The idea behind a message digest is independent of how any particular message digest may be calculated. All message digests share certain features, and the class that abstracts these common features into a single interface is termed an engine. Engines are generally abstract and are always independent of any particular algorithm.

A message digest may be implemented by a particular algorithm, such as MD5 or SHA. An algorithm is generally provided as a concrete class that extends an abstract engine class, completing the definition of the class. However, there may be many classes that provide a particular algorithm; you may have an SHA class that came with your Java platform and you may also have obtained an SHA class from a third party. Both classes should provide the same results, but their internal implementations may be vastly different.

Security providers are the glue that manages the mapping between the engines used by the rest of the security package (such as a message digest), the specific algorithms that are valid for those engines (such as an SHA digest), and the specific implementations of that algorithm/engine pair that might be available to any particular Java virtual machine. The goal of the security provider interface is to allow an easy mechanism where the specific algorithms and their implementations can be easily changed or substituted. The security provider allows us to change the implementation of the SHA digest algorithm that is in use and to introduce a new algorithm to generate a digest.

Hence, a typical programmer only uses the engine classes to perform particular operations. You don't need to worry about the classes that actually perform the computation. The engine classes provide the primary interface to the security package. An administrator, meanwhile, needs to know only the name of the provider class so that she can ensure that the correct provider class is used.

8.1.1 Components of the Architecture

The architecture surrounding all of this has these components:

Engine classes
: These classes come with the Java virtual machine as part of the core API.

Algorithm classes
: At the basic level, there is a set of classes that implement particular algorithms for particular engines. A default set of these classes is provided by the supplier of the Java platform; other third−party organizations (including your own) can supply additional sets of algorithm classes. These classes may implement one or more algorithms for one or more engines; it is not necessary for a set of classes from a particular vendor to implement all possible algorithms or all possible engines. A single algorithm class provides a particular algorithm for a particular engine.

The Provider class
: Each set of algorithm classes from a particular vendor is managed by an instance of the class `Provider`. A provider knows how to map particular algorithms to the actual class that implements the operation.

The Security class
: The `Security` class maintains a list of the provider classes and consults each in turn to see which operations it supports.

In later chapters, we'll look at the individual algorithms and engines of this architecture; for now, we'll discuss the `Provider` and `Security` classes. These two classes together make up the idea of a security provider.

The security providers rely on cooperation between themselves and the rest of the Java security package in order to fulfill their purpose. The details of this cooperation are handled for us `−−` when we use the `MessageDigest` class to generate a digest, for example, it's the responsibility of the `MessageDigest` class to ask the `Security` class which particular class to use to generate the digest. The `Security` class in turn asks each of the providers whether or not they can supply the desired digest.

So a typical program that wants to use the security package does not interact directly with the security provider. Instead, the security provider is transparently useful to the programmer and to the end user. An end user, a system administrator, or a developer can configure the security provider; this is a result of the security provider being based on a set of provider classes. While there is a default provider class, the end user or system administrator can replace the default provider with another class. In addition, a user or programmer can augment the default provider class by adding additional provider classes.

When the security package needs to perform an operation, it constructs a string representing that operation and asks the `Security` class for an object that can perform the operation with the given algorithm. For example, the idea of generating a message digest is represented by a particular engine; its name (i.e., MessageDigest) is the first component in the request to the security provider. There can be many algorithms that can provide a message digest. SHA−1 and MD5 are the two most common, though we'll explore other possibilities when we look in depth at the corresponding classes that handle digests. So the name of the algorithm (e.g., MD5) forms the second component of the string provided to the security class. These components are concatenated into a single string separated by a dot (e.g., MessageDigest.MD5).

Seventeen cryptographic engines are supported by Sun's security providers; there are implementations of at least one algorithm of each engine in one of Sun's providers. The engines and the algorithms implemented by Sun are listed in Table 8−1. Version 1.3 comes with two security providers: Sun, the primary security provider, and SunRsaSign, which implements RSA algorithms. SunJCE is the provider that comes with the Java Cryptography Extension, and SunJSSE is the provider that comes with the Java Secure Sockets Extension. Engines that have an asterisk are JCE engines and have special deployment rules discussed later in this chapter.

Table 8−1. Security Engines and Algorithms

Engine	Algorithm Name	Provider
AlgorithmParameterGenerator	DiffieHellman	SunJCE
AlgorithmParameterGenerator	DSA	Sun
AlgorithmParameters	Blowfish	SunJCE
AlgorithmParameters	DES	SunJCE
AlgorithmParameters	DESede	SunJCE
AlgorithmParameters	DiffieHellman	SunJCE
AlgorithmParameters	DSA	Sun
AlgorithmParameters	PBE	SunJCE
CertificateFactory	X509	Sun
*Cipher	Blowfish	SunJCE
*Cipher	DES	SunJCE
*Cipher	DESede	SunJCE
*Cipher	PBEWithMD5AndDES	SunJCE
*Cipher	PBEWithMD5AndTripleDES	SunJCE
*KeyAgreement	DiffieHellman	SunJCE
KeyFactory	DiffieHellman	SunJCE
KeyFactory	DSA	Sun
KeyFactory	RSA	SunJSSE
KeyFactory	RSA	SunRsaSign
*KeyGenerator	Blowfish	SunJCE
*KeyGenerator	DES	SunJCE
*KeyGenerator	DESede	SunJCE
*KeyGenerator	HmacMD5	SunJCE
*KeyGenerator	HmacSHA1	SunJCE
KeyManagerFactory	SunX509	SunJSSE
KeyPairGenerator	DiffieHellman	SunJCE
KeyPairGenerator	DSA	Sun
KeyPairGenerator	RSA	SunJSSE

KeyPairGenerator	RSA	SunRsaSign
KeyStore	JCEKS	SunJCE
KeyStore	JKS	Sun
KeyStore	PKCS12	SunJSSE
*Mac	HmacMD5	SunJCE
*Mac	HmacSHA1	SunJCE
MessageDigest	MD5	Sun
MessageDigest	SHA	Sun
*SecretKeyFactory	DESede	SunJCE
*SecretKeyFactory	DWES	SunJCE
*SecretKeyFactory	PBEWithMD5AndDES	SunJCE
SecureRandom	SHA1PRNG	Sun
Signature	MD2withRSA	SunJSSE
Signature	MD2withRSA	SunRsaSign
Signature	MD5withRSA	SunJSSE
Signature	MD5withRSA	SunRsaSign
Signature	SHA1withDSA	Sun
Signature	SHA1withRSA	SunJSSE
Signature	SHA1withRSA	SunRsaSign
SSLContext	SSL	SunJSSE
SSLContext	SSLv3	SunJSSE
SSLContext	TLS	SunJSSE
SSLContext	TLSv1	SunJSSE
TrustManagerFactory	SunX509	SunJSSE

The algorithm names in this table are the strings passed to the desired engine class in order for it to find the class implementing the operation. In addition, the security infrastructure also accepts certain alias strings that map an alias to one of these valid strings. For example, the official name of SHA (the Secure Hash Algorithm) for message digests is SHA−1 (to distinguish it from SHA−0, the first such algorithm, which is now obsolete). Internally, security providers can alias one name to another, such that you may specify either SHA or SHA−1 to find a message digest that implements the secure hash algorithm.

8.1.2 Choosing a Security Provider

When the Java virtual machine begins execution, it is responsible for consulting the user's properties in order to determine which security providers should be in place. These properties must be located in the file *$JREHOME/lib/security/java.security*. In Sun's release of 1.3, that file contains these lines (among others):

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
```

These lines tell us that there are at least two provider classes that should be consulted; the first class to be consulted is an instance of the `sun.security.provider.Sun` class and the second class is an instance of the `com.sun.rsajca.Provider` class.

Each provider given in this file must be numbered, starting with 1. If you want to use additional providers, you can edit this file and add those providers at the next numbers. When you obtain JCE, you are told that its provider's class name is `com.sun.crypto.provider.SunJCE`; the classname of the provider in JSSE is `com.sun.net.ssl.internal.ssl.Provider`. Hence, to use these providers you add these lines to the *java.security* file (as we did in Chapter 1):

```
security.provider.3=com.sun.crypto.provider.SunJCE
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
```

Note that there's no reason why the new provider classes had to be added at positions 3 and 4 -- it would have been perfectly acceptable to add the `SunJCE` class as `security.provider.1` if the `sun.security.provider.Sun` class were changed to `security.provider.3` (or, alternately, removed altogether). The `Security` class keeps the instances of the providers in an array so that each class is found at the index specified in the *java.security* file. As long as the providers in the *java.security* file begin with 1 and are numbered consecutively, they may appear in any order.

The order of these properties is significant; when the `Security` class is asked to provide a particular engine and algorithm, it searches the listed providers in order to find the first one that can supply the desired operation. All engine classes use the security class to supply objects. When the message digest engine is asked to provide an object capable of generating SHA message digests, the engine will ask the `Security` class which provider to use. If the first provider in the list can perform SHA message digests, that provider will be used. Otherwise, the second provider is checked, and so on, until there are no providers left (and an exception is thrown) or until a provider that implements the desired operation is found. Hence, the number that follows the `security.provider` string indicates the order in which providers will be searched for particular implementations.

Note that the classes that are listed in this manner must be installed into the system classpath. The security infrastructure will use only the system class loader to locate these classes, which is why they are usually installed as an extension in *$JREHOME/lib/ext*. If you write your own security provider (as we do in the next section), that provider must either be installed programatically or the classes that encompass the provider must be installed as an extension.

For end users and administrators, that's all there is to adding new security providers. For developers, there is also a programmatic way in which a security provider may be added; we'll explore that when we discuss the interface of the `Security` class. But as we mentioned earlier, the programmatic interface provided by the two classes we're about to discuss is not often needed; you'd need it only if you wanted to supply your own security provider or if you wanted to inspect or set programmatically the list of existing providers. Otherwise, the classes are interesting only because they are used by the engine classes we'll begin to examine in the next chapter.

8.2 The Provider Class

The first class we'll examine in depth is the `Provider` class (`java.security.Provider`).

public abstract class Provider extends Properties

> This class forms the basis of the security provider architecture. There is normally a standard subclass that implements a default security feature set; other classes can be installed to implement other security algorithms.

In the core Java API, the `Provider` class is abstract and there are no classes in the core Java API that extend the `Provider` class. The default provider class that comes with Sun's implementation of Java is the class `Sun` in the `sun.security.provider` package. However, since this class is in the `sun` package, there's no guarantee that it will be available with every implementation of the Java virtual machine; other implementations may supply a different provider.

In theory, this should not matter. The concepts of the security package will work according to the specification as long as the Java implementation provides an appropriate provider class and appropriate classes to perform the operations a Java program will expect. The exact set of classes a particular program may expect will depend, of course, on the program. In the next section, we'll discuss how different implementations of the `Provider` class may be loaded and used during the execution of the virtual machine.

8.2.1 Using the Provider Class

The `Provider` class is seldom used directly by a programmer. This class does contain a number of useful miscellaneous methods we'll review here; these methods are generally informational and would be used accordingly:

public String getName( )
: Return the name of the provider.

public double getVersion( )
: Return the version number of the provider.

public String getInfo( )
: Return the info string of the provider.

public String toString( )
: Return the string specifying the provider; this is typically the provider's name concatenated with the provider's version number.

As an extension of the `Properties` class, the `Provider` class also shares its public interface. The `Provider` class overrides three of those methods:

public synchronized void clear( )
: If permission is granted, clear out all entries from the provider.

public synchronized Object put(Object key, Object value)
: If permission is granted, add the given property, keyed off the given key.

public synchronized Object remove(Object key)
: If permission is granted, remove the object associated with the given key.

These methods are overridden so that the `Provider` class can ensure that the program has sufficient permissions to perform the operation. These methods call the `checkSecurityAccess( )` method of the security manager, which in turn uses the access controller to determine if the current classes have been granted a `SecurityPermission` with the appropriate name. The string used for the `clear( )` method is `clearProviderProperties.name`; for the `put( )` method, it is `putProviderProperty.name`; and for the `remove( )` method, it is `removeProviderProperty.name`. The name in these strings is replaced by the name of the provider itself (e.g., `Sun` for the standard security provider).

Since the interface to this class is simple, we won't actually show how it is used, although we will use some of these methods later in this chapter. Note also that there is no public constructor for the `Provider` class −− a provider can only be constructed under special circumstances we'll discuss later.

8.2.2 Implementing the Provider Class

If you're going to provide your own set of classes to perform security operations, you must extend the `Provider` class and register that class with the security infrastructure. In this section, we'll explore how to do that. Most of the time, of course, you will not implement your own `Provider` class −− you'll just use the default providers or perhaps install a third−party provider using the techniques that we explore in the next section.

Although the `Provider` class is abstract, none of its methods are abstract. This means that implementing a provider is, at first blush, simple: all you need do is subclass the `Provider` class and provide an appropriate constructor. The subclass must provide a constructor since there is no default constructor within the `Provider` class. The only constructor available to us is:

protected Provider(String name, double version, String info)
Construct a provider with the given name, version number, and information string.

Hence, the basic implementation of a security provider is:

```
public class XYZProvider extends Provider {
        public XYZProvider(  ) {
                super("XYZ", 1.0, "XYZ Security Provider v1.0");
        }
}
```

Here we're defining the skeleton of a provider that is going to provide certain facilities based on various algorithms of the XYZ Corporation. Throughout the remainder of this book, we'll be developing the classes that apply to the XYZ's cryptographic methods, but they will be examples only −− they lack the rigorous mathematical properties that these algorithms must have. In practice, you might choose to implement algorithms that correspond to the RSA algorithms for the cryptographic engines. Good examples of implementing cryptographic algorithms can be found in Jonathan Knudsen's *Java Cryptography* (O'Reilly).

Note we used a default constructor in this class rather than providing a constructor similar to the one found in the `Provider` class itself. The reason for this has to do with the way providers are constructed, which we discuss at the end of this section. When you write a provider, it must provide a constructor with no arguments.

This is a complete, albeit useless, implementation of a provider. In order to add some functionality to our provider, we must put some associations into the provider. The associations will perform the mapping that we mentioned earlier; it is necessary for the provider to map the name of an engine and algorithm with the name of a class that implements that operation. This is why the `Provider` class itself is a subclass of the `Properties` class −− so that we can make each of those associations into a property.

The operations that our provider will be consulted about are listed in Table 8−2. In this example, we're going to be providing an SHA algorithm for performing message digests since that would be needed as part of the signature generation algorithm we want to implement. There's no absolute requirement for this; we could have depended on the default Sun security provider to supply this algorithm for us. On the other hand, there's no guarantee that the default security provider will be in place when our security provider is installed, so it's a good idea for a provider to include all the algorithms it will need.

Table 8−2. Properties Included by Our Sample Provider

Property	Corresponding Class
`KeyGenerator.XOR`	`javasec.samples.ch09.XORKeyGenerator`
`KeyPairGenerator.XYZ`	`javasec.samples.ch09.XYZKeyPairGenerator`
`KeyFactory.XYZ`	`javasec.samples.ch09.XYZKeyFactory`
`MessageDigest.XYZ`	`javasec.samples.ch11.XYZMessageDigest`
`Signature.XYZwithSHA`	`javasec.samples.ch12.XYZSignature`
`Cipher.XOR`	`javasec.samples.ch13.XORCipher`
`KeyManagerFactory.XYZ`	`javasec.samples.ch14.SSLKeyManagerFactory`

In order to make the associations from this table, then, our `XYZProvider` class needs to look like this:

```
package javasec.samples.ch08;

import java.security.*;

public class XYZProvider extends Provider {
    public XYZProvider(  ) {
        super("XYZ", 1.0, "XYZ Security Provider v1.0");
        // These are examples we'll demonstrate throughout the next
        // chapters.
        put("KeyGenerator.XOR",
                      "javasec.samples.ch09.XORKeyGenerator");
        put("KeyPairGenerator.XYZ",
                      "javasec.samples.ch09.XYZKeyPairGenerator");
        put("KeyFactory.XYZ", "javasec.samples.ch09.XYZKeyFactory");
        put("MessageDigest.XYZ",
                      "javasec.samples.ch11.XYZMessageDigest");
        put("Signature.XYZwithSHA",
                      "javasec.samples.ch12.XYZSignature");
        put("Cipher.XOR", "javasec.samples.ch13.XORCipher");
        put("KeyManagerFactory.XYZ",
                      "javasec.samples.ch14.SSLKeyManagerFactory");

        // Now include any aliases
        put("Alg.Alias.MessageDigest.SHA-1", "SHA");
    }

    public static final synchronized void verifyForJCE(  ) {
        // See Appendix E for more details
        throw new SecurityException("Can't verify for JCE");
    }
}
```

The only properties a provider is required to put into its property list are the properties that match the engine name and algorithm pair with the class that implements that operation. In this example, that's handled with the first four calls to the `put( )` method (but remember too that the provider can implement as few or as many operations as it wants to; it needn't implement more than a single engine with one algorithm, or it can implement dozens of engine/algorithm pairs). Note that the class name is the fully qualified package name of the class.

The provider also has the opportunity to set any other properties that it wants to use. If the provider wants to set aliases (as we've done with the final call to the `put( )` method), it's free to do so. The last code line of our example shows the syntax to do this; it means that the string SHA-1 can be used instead of the string SHA when searching for a message digest class.

A provider can set any other arbitrary properties that it wants as well. For instance, a provider class could set this property:

```
put("NativeImplementation", "false");
```

if it wanted the classes that use the provider to be able to determine if this particular XYZ implementation uses native methods.[1] It can also use the convention that certain properties are preceded with the word `Alg` and contain the algorithm name, like this:

> [1] RSA algorithms sometimes use native methods because there are existing implementations of them that are written in C and have gone through an extensive quality acceptance test that many commercial sites have a level of confidence in. However, as Java has become more widely used, this is now the exception.

```
put("Alg.NativeImplementation.XYZ", "false");
```

There's no advantage to setting any additional properties -- nothing in the core SDK will use them. They can be set to make the classes that accompany your provider class easier to write -- for example, your `XYZSignature` class might want to inquire which particular providers have a native method implementation of the XYZ algorithm. Whatever information you put into your provider and how your accompanying classes use that information is a design detail that is completely up to you. The `Security` class will help you manage the information in these properties; this relationship to the `Security` class is the reason we used a string value for the `NativeImplementation` property rather than a `Boolean` value.

There's one more nonpublic method of the `Provider` class that is used by the security API:

static Provider loadProvider(String className)
: Instantiate a provider that has as its type the given class. This method is provided mostly for convenience -- it simply loads the given class and instantiates it. However, this method also ensures that the loaded class is an instance of the `Provider` class.

This method creates an instance of a provider. The importance of this method stems from how it performs its task: it creates the instance of the provider object by calling the `newInstance( )` method of the `Class` class. In order for that operation to succeed, the provider class must therefore have a default constructor -- that is, a constructor that requires no arguments. This is why in our example we provided such a constructor and had the constructor hardwire the name, version number, and information string. We could have provided an additional constructor that accepts those values as parameters, but it would never be called since the only way in which the virtual machine uses providers is to load them via this method.

8.2.3 Deploying the Provider Class

If you want to write your own provider, you must consider how that provider will be deployed. That's because certain engines (those defined wholly within JCE) are required to be deployed in a special way. Although JCE is exportable, there are restrictions on how new implementations of JCE engines can be introduced. JCE enforces those restrictions by requiring that a JCE−compatible provider be deployed as a signed jar file with other special information.

Hence, if you want to deploy a provider with a JCE engine, you must follow the instructions at http://java.sun.com/products/jce/doc/guide/HowToImplAProvider.html. In simple terms, you must apply to Sun or to IBM for a special code−signing certificate. This entails submitting a request to a special certificate authority and providing that authority with hard copy documentation establishing your identity. Once you receive the certificate, you package your provider (and its dependent classes) as a jar file, which you sign with this certificate. There are some coding requirements for your JCE engine classes as well, which are handled in the `verifyForJCE( )` method. A valid implementation of that method requires a lot of the security classes that we haven't examined yet, so we've just provided a stub here. We'll discuss how to implement that method in Appendix E.

There is no such restriction for deploying a security provider that implements non−JCE engines. Chapter 8 flags the engines that have such a restriction with an asterisk. Also note that JCE security providers are allowed to supply implementations for non−JCE engines; Sun's JCE provider does just that when it supplies a key pair generator. You can have a mixed security provider and deploy it either as a specially−signed JCE provider or deploy it as a simple jar file (or, even more simply, as a set of classes). In the latter case, the non−JCE engines will still function, and an exception will be thrown if you attempt to retrieve a JCE engine.

8.3 The Security Class

In this section, we'll look into how the Java VM locates the security provider(s) we want to use. The `Security` class (`java.security.Security`) is responsible for managing the set of provider classes that a Java program can use and forms the last link in the architecture of the security provider. This class is final, and all its methods are static (except for its constructor, which is private). Like the `System` and `Math` classes, then, the `Security` class can never be created or subclassed; it exists simply to provide a placeholder for methods that deal with the `java.security` package.

Earlier, we explained how to add entries to the *java.security* file to add new providers to the security architecture. The same feat can be accomplished programmatically via these methods of the `Security` class:

public static int addProvider(Provider provider)
: Add a new provider into the list of providers. The provider is added to the end of the internal array of providers.

public static int insertProviderAt(Provider provider, int position)

Add a new provider into the internal array of providers. The provider is added at the specified position; other providers have their index changed if necessary to make room for this provider. Position counting begins at 1.

The notion that these classes are kept in an indexed array is important; when the `Security` class is asked to provide a particular algorithm for an operation, the array is searched sequentially for a provider that can supply the requested algorithm for the requested operation.

As an example, let's assume the existence of an SDO security provider. This class comes with a set of classes to perform generation of key pairs, and it can generate key pairs according to two algorithms: DSA and XYZ. The `SDO` class, according to an entry added to the *java.security* file, has been added at position 2. Additionally, let's say that our Java program has installed an additional provider class at position 3 called JRA that can generate key pairs and digital signatures according to a single algorithm known as Foo. This leaves us with the set of provider classes listed in Table 8−3.

Table 8−3. Sample Security Providers

Sun Provider	SDO Provider	JRA Provider
Signature Engines DSA	Signature Engines XYZ DSA	Signature Engines Foo
Message Digest Engines MD5	Message Digest Engines XYZ SHA	Message Digest Engines None
Key Pair Engines DSA	Key Pair Engines XYZ DSA	Key Pair Engines Foo

Now when our Java program needs to generate a key pair, the security provider is consulted as to which classes will implement the key pair generation we want. If we need to generate a DSA key, the security provider returns to us a class associated with the Sun provider class since the Sun provider, at position 1, is the first class that says that it can perform DSA key generation. If we had reversed the order of indices in the *java.security* file so that the Sun provider was at position 2 and the SDO provider was at position 1, a class associated with the SDO provider would have been returned instead. Similarly, when we request a Foo key pair, a class associated with the JRA provider is returned to us, regardless of what index it occurs at, since that is the only provider class that knows how to perform Foo key generation.

Remember that this is a two−step process. The security class receives a string (like `KeyPairGenerator.DSA`) and locates a class that provides that service (such as `sun.security.provider.Sun`). The `Sun` class, as a provider class, does not actually know how to generate keys (or do anything else) −− it only knows what classes in the Sun security package know how to generate keys. Then the security class must ask the provider itself for the name of the class that actually implements the desired operation. That process is handled by an internal method of the `Security` class −− we'll use that method implicitly over the next few chapters when we retrieve objects that implement a particular engine and algorithm. Before we do that, though, we'll finish looking at the interface of the

`Security` class.

There are a number of other methods in the `Security` class that provide basic information about the configuration of the security provider:

public static void removeProvider(String name)

Remove the named provider from the list of provider classes. The remaining providers move up in the array of providers if necessary. If the named provider is not in the list, this method silently returns (i.e., no exception is thrown).

public static Provider[] getProviders( )

Return a copy of the array of providers on which the `Security` class operates. Note that this is a copy of the array; reordering its elements has no effect on the `Security` class.

public static Provider getProvider(String name)

Return the provider with the given name. If the named provider is not in the list held by the `Security` class, this method returns `null`.

public static String getProperty(String key)

Get the property of the `Security` class with the associated key. The properties held in the `Security` class are the properties that were read from the *java.security* file. In typical usage, one of the properties is `security.provider.1` (as well as any other providers listed in the *java.security* file). Note, however, that properties of this sort may not reflect the actual order of the provider classes: when the `addProvider( )`, `insertProviderAt( )`, and `removeProvider( )` methods are called, the order of the providers changes. These changes are not reflected in the internal property list.

The *java.security* file has a number of other properties within it; these other properties may also be retrieved via this method.

public static void setProperty(String property, String value)

Set the given property to the given value.

public static String getAlgorithmProperty(String algName, String propName)

Search all the providers for a property in the form `Alg.propName.algName` and return the first match it finds. For example, if a provider had set the `Alg.NativeImplementation.XYZ` property to the string "false," a call to `getAlgorithmName("XYZ", "NativeImplementation")` returns the string "false" (which is why earlier we used a string value in the provider class).

Here's a simple example, then, of how to see a list of all the security providers in a particular virtual machine:

```
package javasec.samples.ch08;

import java.security.*;
import java.util.*;

public class ExamineSecurity {
    public static void main(String args[]) {
        try {
            Provider p[] = Security.getProviders(  );
            for (int i = 0; i < p.length; i++) {
                System.out.println(p[i]);
                for (Enumeration e = p[i].keys(); e.hasMoreElements(  );)
                    System.out.println("\t" + e.nextElement(  ));
            }
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

If we run this program with the 1.3 default security providers, we get the following output (although we've removed some lines for clarity):

```
SUN version 1.2
        Alg.Alias.KeyFactory.1.2.840.10040.4.1
        Alg.Alias.Signature.1.2.840.10040.4.3
        Alg.Alias.KeyPairGenerator.OID.1.2.840.10040.4.1
        AlgorithmParameterGenerator.DSA
        Alg.Alias.KeyPairGenerator.1.3.14.3.2.12
        Alg.Alias.Signature.SHA/DSA
        Alg.Alias.Signature.1.3.14.3.2.13
        SecureRandom.SHA1PRNG
        Alg.Alias.CertificateFactory.X.509
        Alg.Alias.Signature.DSS
        Signature.SHA1withDSA
        AlgorithmParameters.DSA
        MessageDigest.SHA
        CertificateFactory.X509
        Alg.Alias.AlgorithmParameters.1.3.14.3.2.12
        Alg.Alias.Signature.DSA
        KeyStore.JKS
        Alg.Alias.Signature.DSAWithSHA1
        MessageDigest.MD5
        KeyPairGenerator.DSA
        Alg.Alias.Signature.SHAwithDSA
        Alg.Alias.Signature.OID.1.2.840.10040.4.3
        Alg.Alias.Signature.SHA1/DSA
        Alg.Alias.KeyPairGenerator.1.2.840.10040.4.1
        Alg.Alias.MessageDigest.SHA-1
        Alg.Alias.MessageDigest.SHA1
        Alg.Alias.AlgorithmParameters.1.2.840.10040.4.1
        KeyPairGenerator.DSA KeySize
        KeyFactory.DSA
        Alg.Alias.Signature.1.3.14.3.2.27
        Alg.Alias.Signature.SHA-1/DSA
        AlgorithmParameterGenerator.DSA KeySize
SunRsaSign version 1.0
        KeyFactory.RSA
        Signature.MD5withRSA
        Signature.SHA1withRSA
        Signature.MD2withRSA
        KeyPairGenerator.RSA
```

Two things are readily apparent from this example. First, the strings that contain only an engine name and an algorithm implement the expected operations that we listed in Table 8−1. Second, as we mentioned in the section on the `Provider` class, security providers often leverage the fact that the `Provider` class is a subclass of the `Properties` class to provide properties that may make sense only to other classes that are part of the provider package. Hence, the signature algorithm `1.3.14.3.2.13` may make sense to one of the classes in the Sun security provider, but it is not a string that will necessarily make sense to other developers. In fact, those aliases −− including the ones that are prefaced by OID −− do have meanings within the cryptography standards world, but for our purposes we'll stick with the standard algorithm names that we listed earlier.

8.3.1 The Security Class and the Security Manager

Some of the public methods of the `Security` class call the `checkSecurityAccess( )` method of the security manager. This gives the security manager the opportunity to intervene before an untrusted class affects the security policy of the virtual machine.

Recall that the `checkSecurityAccess( )` method accepts a single string parameter. In the case of the methods in the `Security` class, the call that is made looks like this:

```
public static synchronized int
       insertProviderAt(Provider provider, int position) {
        SecurityManager sec = System.getSecurityManager(  );
        if (sec != null)
                sec.checkSecurityAccess("insertProvider."+provider.getName(  ));
        ... continue to find the provider ...
}
```

So a program that wants to install the XYZ security provider must have been granted the `SecurityPermission` named "insertProvider.XYZ". The methods of the security class that require a security permission and the names of the permission they require are listed in Table 8−4.

Table 8−4. Security Checks of the Security Class

Method	Parameter
`insertProviderAt(  )`	`insertProvider. + provider.getName(  )`
`removeProvider(  )`	`removeProvider. + provider.getName(  )`
`getProperty(  )`	`getProperty. + key`
`setProperty(  )`	

`setProperty. + key`

8.4 The Architecture of Engine Classes

In the next few chapters, we'll discuss the engine classes that are part of the core Java API and the security extensions. All engine classes share a similar architecture that we'll discuss here.

Most programmers are only interested in using the engine classes to perform their desired operation; each engine class has a public interface that defines the operations the engine can perform. None of this is unusual: it is the basis of programming in Java.

However, the engine classes are designed so that users can employ third−party security providers (using the architecture we've just examined). For programmers who are interested in writing such providers, the engine classes have an additional interface called the security provider interface (SPI). The SPI is a set of abstract methods that a particular engine must implement in order to fulfill its contract of providing a particular operation.

For most cryptographic engines, the SPI is unrelated to the engine itself; the `CertificateFactorySpi` class extends the `Object` class. For historical reasons, this is not true of three engine classes: the `KeyPairGeneratorSpi` , `MessageDigestSpi`, and `SignatureSpi` classes each extends an engine class (the `KeyPairGenerator` class and so on). However, this difference in class hierarchies has no practical effect on developers.

Hence, if you want to implement a security provider, you extend the SPI of each engine that you want to provide. This allows a developer to request a particular engine and receive the correct class according to the following algorithm:

1. The programmer requests an instance of a particular engine that implements a particular algorithm. Engine classes never have public constructors; instead, every engine has a `getInstance( )` method that takes the name of the desired algorithm as an argument and returns an instance of the appropriate class.
2. The `Security` class is asked to consult its list of providers and provide the appropriate instance. For example, when the `getInstance( )` method of the `MessageDigest` class is called, the `Security` class may determine that the appropriate provider class is called `com.xyz.XYZMessageDigest`.
3. If the retrieved class does not extend the appropriate SPI (e.g., `java.security.MessageDigestSpi` in this case), a `NoSuchAlgorithmException` is generated.
4. An instance of the retrieved class is created and returned to the `getInstance( )` method (which in turn returns it to the developer).

We'll show examples of this in later chapters for particular engines.

8.5 Comparison with Previous Releases

There are no changes in the security provider infrastructure between 1.2 and 1.3. However, the `SunRsaSign` security provider is available only with 1.3; although 1.2 defines interfaces for RSA keys, you must obtain a third−party security provider to use them. The `SunJSSE` and `SunJCE` security providers may be installed into 1.3.

The security provider infrastructure works essentially the same in 1.1, but 1.1 supplies fewer engines. In 1.1, there are only engines to perform key pair generation, message digests, and digital signatures. There are no

SPI classes in 1.1, so to implement an engine you extend the engine class directly (1.2 is backward−compatible with these classes, which is why the class hierarchy differs for these engines). If you must provide a engine that can be used in both 1.1 and 1.2, you should extend the engine class rather than the SPI.

In 1.1, the `Provider` class does not override the `clear( )` , `put( )`, and `remove( )` methods. In the `Security` class, certain methods still call the security manager to see if their operation should continue, but the string passed to the security manager is always simply the string "java." In addition, the `getProviders( )` , `getProvider( )`, and `getProperty( )` methods also perform this check in 1.1.

8.6 Summary

In this chapter, we've explored the architecture that forms the basis of the Java security API. This architecture is based on the `Security` and `Provider` classes, which together form a set of mappings that allow the security API to determine dynamically the set of classes it should use to implement certain operations.

Implementing a provider is trivial, but implementing the set of classes that must accompany a provider is much harder. We've shown a simple provider class in this chapter. Although we'll show the engine classes in the next few chapters, the mathematics behind designing and implementing a successful cryptographic algorithm are beyond the scope of this book. However, this architecture also allows users and administrators to buy or download third−party implementations of the security architecture and plug those implementations seamlessly into the Java virtual machine; a partial list of available third−party implementations appears on Sun's web site (see Appendix B, for more information).

In the next few chapters, we'll examine the specifics of the engine classes −− that is, the operations −− that this security provider architecture makes possible. In those chapters, we'll see how the engines are used and the benefits each engine provides.

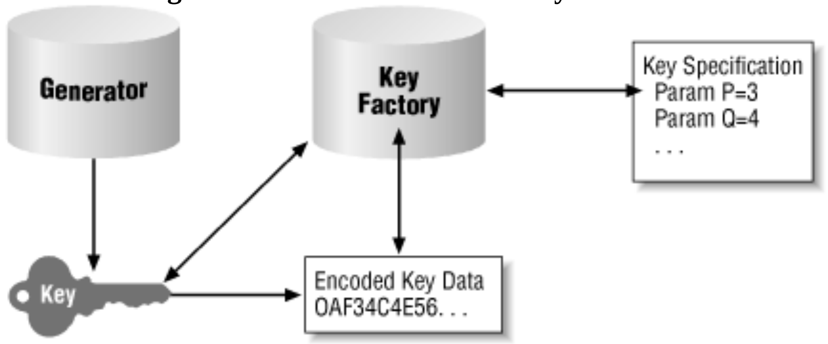
Chapter 9. Keys and Certificates

In this chapter, we discuss the classes in the Java security package that handle keys and certificates. Keys are a necessary component of many cryptographic algorithms −− in particular, keys are required to create and verify digital signatures or to perform encryption. Different algorithms require different keys. There are two general types of keys: asymmetric and symmetric. Asymmetric keys come in two types as well, public and private. A public key and a private key are related and are referred to as a key pair. Symmetric keys are also called secret keys.

We also cover the implementation of certificates in this chapter. Certificates are used to authenticate public keys; when public keys are transmitted electronically, they are often embedded within certificates. The core Java API comes with the necessary classes to handle public and private keys and their certificates. The classes necessary to handle secret keys come only with JCE.

Keys and certificates are normally associated with some person or organization, and the way in which keys are stored, transmitted, and shared is an important topic in the security package. Management of keys is left for the next chapter, however; right now, we're just concerned about the APIs that implement keys and certificates. In this chapter, we'll show how a programmer interacts with keys and certificates as well as how you might implement your own versions of each. The classes and engines we discuss in this chapter are outlined in Figure 9−1. There are two engines that operate on keys:

Figure 9−1. The interaction of key classes



- A generator class creates keys from scratch. With no input (or, possibly, input to initialize it to a certain state), the generator can produce one or more keys. Symmetric keys are generated by the `KeyGenerator` class while asymmetric key pairs are generated by the `KeyPairGenerator` class.
- The `KeyFactory` class translates between key objects and their external representations, which may be either a byte array or a key specification.

There are a number of classes and interfaces relating to Figure 9−1; in addition to the engine classes themselves, there are several classes and interfaces that represent the key objects and the key specifications (the encoded key data is always an array of bytes). In an effort to provide the complete story, we'll delve into the details of all of these classes; for the most part, however, the important operations that most developers will need are:

- The ability to create new keys from scratch using the key pair generator or the key generator.
- The ability to export a key, either as a parameter specification or as a set of bytes, and the corresponding ability to import that data in order to create a key.

This means that, for the most part, the data objects we explore in this chapter −− the `Key` classes and interfaces as well as the various `KeySpec` classes (key specification classes) −− can be treated by most programmers as opaque objects. We'll show their complete interface (which you might be curious about and which is absolutely needed if you're writing your own security provider), but we'll try not to lose sight of the two goals of this chapter.

9.1 Keys

Let's start with the various classes that support the notion of keys within Java.

9.1.1 The Key Interface

The concept of a key is modeled by the `Key` interface (`java.security.Key`):

public interface Key extends Serializable

Model the concept of a single key. Because keys must be transferred to and from various entities, all keys must be serializable.

As we discussed in Chapter 8, there might be several algorithms available for generating (and understanding) keys, depending on the particular security providers that are installed in the virtual machine. Hence, the first thing a key needs to be able to tell us is what algorithm generated it:

public String getAlgorithm( )

Return a string describing the algorithm used to generate this key; this string should be the name of a standard key generation algorithm.

When a key is transferred between two parties, it is usually encoded as a series of bytes; this encoding must follow a format defined for the type of key. Keys are not required to support encoding −− in which case the format of the data transferred between the two parties in a key exchange is either obvious (e.g., simply the serialized data of the key) or specific to a particular implementation. Keys tell us the format they use for encoding their output with this method:

public String getFormat( )

Return a string describing the format of the encoding the key supports.

The encoded data of the key itself is produced by this method:

public byte[] getEncoded( )

Return the bytes that make up the particular key in the encoding format the key supports. The encoded bytes are the external representation of the key in binary format.

Those are the only methods that a key is guaranteed to implement (other than methods of the `Object` class, of course; most implementations of keys override many of those methods). In particular, you'll note that there is nothing in the key interface that says anything about decoding a key. We'll say more about that later.

9.1.2 Asymmetric Keys

Asymmetric keys are the more popular type of key. These keys come in pairs; hence the core Java API contains these two additional interfaces:

public interface PublicKey extends Key *public interface PrivateKey extends Key*

These interfaces contain no additional methods; they are used simply for type convenience. A class that implements the `PublicKey` interface identifies itself as a public key, but it contains no public methods different from any other key.

The security providers that come with Sun's implementation of Java provide two types of asymmetric keys: DSA and RSA. JCE provides an additional type of asymmetric key: Diffie−Hellman. Each of these have their own interface that allows you to determine fundamental information about the key. For most programmers, however, keys are opaque objects, and the algorithm−specific features of keys are not needed (except in certain cases when you need to transfer keys).

9.1.2.1 DSA keys

These interfaces allow you to retrieve the DSA algorithm−specific parameters P, Q, and G that are used to generate the keys. Knowledge of these variables is abstracted into the `DSAParams` interface (`java.security.interfaces.DSAParams`):

```
public interface DSAParams {
        public BigInteger getP(  );
        public BigInteger getQ(  );
        public BigInteger getG(  );
}
```

Keys that are generated by DSA will typically implement the `DSAKey` interface (`java.security.interfaces.DSAKey`):

public interface DSAKey
: Provide DSA−specific information about a key.

Implementing this interface serves two purposes. First, it allows the programmer to determine if the key is a DSA key by checking its type. The second purpose is to allow the programmer to access the DSA parameters using this method in the `DSAKey` interface:

public DSAParams getParams( )
: Return the DSA parameters associated with this key.

These methods and interfaces allow us to do specific key manipulation like this:

```
public void printKey(Key k) {
    if (k instanceof DSAKey) {
        System.out.println("key is DSA");
        System.out.println("P value is " +
                        ((DSAKey) k).getParams().getP(  ));
    }
    else System.out.println("key is not DSA");
}
```

The idea of a DSA key is extended even further by these two interfaces (both of which are in the `java.security.interfaces` package):

public interface DSAPrivateKey extends DSAKey
public interface DSAPublicKey extends DSAKey

These interfaces allow the programmer to retrieve the additional key−specific values (known as Y for public keys and X for private keys in the DSA algorithm):

```
public void printKey(DSAKey k) {
    if (k instanceof DSAPublicKey)
        System.out.println("Public key value is " +
                                        ((DSAPublicKey) k).getY(  ));
    else if (k instanceof DSAPrivateKey)
        System.out.println("Private key value is " +
                                        ((DSAPrivateKey) k).getX(  ));
    else System.out.println("Bad key implementation");
}
```

DSA keys are often used in the Java world (and elsewhere in cryptography), and if you know you're dealing with DSA keys, these interfaces can be very useful. In particular, if you're writing a security provider that provides an implementation of DSA keys, you should ensure that you implement all of these interfaces correctly.

9.1.2.2 RSA keys

Unlike their DSA counterparts, RSA keys do not share a common type. However, the following interfaces exist for particular RSA keys (within the `java.security.interfaces` package):

public interface RSAPrivateKey extends PrivateKey
public interface RSAPrivateKeyCrt extends RSAPrivateKey
public interface RSAPublicKey extends PublicKey

These interfaces define keys suitable for use in RSA algorithms.

These interfaces allow you to retrieve the parameters used to create the RSA key. In particular, the RSA public key interface has methods to return its modulus and public exponent while the private key has methods to return its modulus and private exponent. The `RSAPrivateKeyCrt` interface defines additional methods to return its prime values (known as P and Q) and their exponents.

9.1.2.3 Diffie−Hellman keys

Diffie−Hellman keys are used in secret key agreement, a topic we address in the next chapter. That algorithm is available only within JCE, and Diffie−Hellman key interfaces are defined in the `javax.crypto.interfaces` package:

public interface DHKey
public interface DHPublicKey extends DHKey, PublicKey
public interface DHPrivateKey extends DHKey, PrivateKey

This set of interfaces defines keys suitable for use in Diffie−Hellman algorithms.

Diffie−Hellman parameters are encoded into the `javax.crypto.spec.DHParameterSpec` class, which includes the Diffie−Hellman parameters G, L, and P. The Diffie−Hellman public key interface includes the parameter Y, and the Diffie−Hellman private key interface includes the parameter X.

9.1.2.4 The KeyPair class

There is one additional class, the `KeyPair` class (`java.security.KeyPair`), that extends the abstraction of asymmetric keys:

public final class KeyPair

Model a data object that contains a public key and a private key.

The `KeyPair` class is a very simple data structure class containing two pieces of information: a public key and a private key. When we need to generate our own keys (which we'll do next), we'll need to generate both the public and private key at once. This object will contain both of the necessary keys. If you're not interested in generating your own keys, this class may be ignored.

The `KeyPair` class contains only two methods:

public PublicKey getPublic( )
public PrivateKey getPrivate( )
Return the desired key from the key pair.

A key pair object is instantiated through a single constructor:

public KeyPair(PublicKey pub, PrivateKey priv)
Create a key pair object, initializing each member of the pair.

In theory, a key pair should not be initialized without both members of the pair being present; there is nothing, however, that prevents us from passing `null` as one of the keys. Similarly, there are no security provisions within the `KeyPair` class that prevent the private key from being accessed -- no calls to the security manager are made when the `getPrivate( )` method is invoked. Hence the `KeyPair` class should be used with caution.

9.1.3 Symmetric Keys

Symmetric (or secret) keys are used only within JCE, where they are defined by the `SecretKey` interface (`javax.crypto.SecretKey`):

public interface SecretKey extends Key
Identify a class as being a symmetric key.

Secret keys have no specific identifying information; this interface is empty and is used only for type identification. There are many types of secret keys used within JCE: Blowfish, DES, DESede, HmacMD5, HmacSHA1, PBEWithMD5AndDES, and PBEWithMD5AndTripleDES. Unlike asymmetric keys, however, these different key types do not have their own interfaces.

9.2 Generating Keys

Java's security API provides two standard engines to generate keys: one to generate a pair of asymmetric keys and one to generate a secret key.

9.2.1 The KeyPairGenerator Class

Generation of public and private keys is provided by the `KeyPairGenerator` class (`java.security.KeyPairGenerator`):

public abstract class KeyPairGenerator extends KeyPairGeneratorSpi
Generate and provide information about public/private key pairs.

Generating a key pair is a very time−consuming operation. Fortunately, it does not need to be performed often; much of the time, we obtain keys from a key management system rather than generating them. However, when we establish our own key management system in the next chapter, we'll need to use this class; it is often easier to generate your own keys from scratch rather than use a key management system as well.

9.2.1.1 Using the KeyPairGenerator class

Like all engine classes, the `KeyPairGenerator` is an abstract class for which there is no implementation in the core API. However, it is possible to retrieve instances of the `KeyPairGenerator` class via these methods:

public static KeyPairGenerator getInstance(String algorithm)
public static KeyPairGenerator getInstance(String algorithm, String provider)

> Find the implementation of the engine that generates key pairs with the named algorithm. The algorithm should be one of the standard API algorithm names; if an appropriate implementation cannot be found, this method throws a `NoSuchAlgorithmException`.
>
> The first format of this method searches all available providers according to the rules we outlined in Chapter 8. The second method searches only the named provider, throwing a `NoSuchProviderException` if that provider has not been loaded.

These methods search the providers that have been registered with the security provider interface for a key pair generator that supports the named algorithm. The names supported by the standard Sun security provider are "DSA" and "RSA"; the JCE security provider also supports the name "Diffie−Hellman." The JSSE security provider supplies its own implementation of RSA keys.

Once we have the key pair generator, we can invoke any of the following methods on it:

public String getAlgorithm( )

> Return the name of the algorithm that this key pair generator implements (e.g., DSA).

public void initialize(int strength)
public abstract void initialize(int strength, SecureRandom random)

> Initialize the key pair generator to generate keys of the given strength. The idea of strength is common among key pair generator algorithms; typically it means the number of bits that are used as input to the engine to calculate the key pair, but the actual meaning may vary between algorithms.
>
> Most key algorithms restrict the values that are valid for `strength`. In the case of DSA and Diffie−Hellman, the strength must be between 512 and 1024 and it must be a multiple of 64. For RSA, the strength must be any number between 512 and 2048. If an invalid number is passed for `strength`, an `InvalidParameterException` will be thrown.
>
> Key pairs require a random number generator to assist them. You may specify a particular random number generator if desired; otherwise, a default random number generator (an instance of the `SecureRandom` class) is used.

public void initialize(AlgorithmParameterSpec params)
public void initialize(AlgorithmParameterSpec params, SecureRandom random)

> Initialize the key pair generator using the given parameter specification (which we'll discuss a little later). By default, the first method simply calls the second method with a default instance of the

`SecureRandom` class.

public abstract KeyPair generateKeyPair( )
public final KeyPair genKeyPair( )
Generate a key pair using the initialization parameters previously specified. A `KeyPairGenerator` object can repeatedly generate key pairs by calling one of these methods; each new call generates a new key pair. The `genKeyPair( )` method simply calls the `generateKeyPair( )` method.

Using these methods, generating a pair of keys is very straightforward:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
kpg.initialize(512);
KeyPair kp = kpg.generateKeyPair(  );
```

According to the Java documentation, you are allowed to generate a key pair without initializing the generator; in this situation, a default strength and random number generator are to be used. However, this feature has not always worked: a `NullPointerException` is sometimes thrown from within the `generateKeyPair( )` method. Since it is possible that third−party providers may behave similarly, it is always best to initialize the key pair generator.

We'll show what to do with these keys in the next chapter when we discuss the topic of key management.

9.2.1.2 Generating DSA keys

The abstraction provided by the key pair generator is usually all we need to generate keys. However, sometimes the particular algorithm needs additional information to generate a key pair. When a DSA key pair is generated, default values for P, Q, and G are used; in the Sun security provider, these values are precomputed to support strength values of 512 and 1024. Precomputing these values greatly reduces the time required to calculate a DSA key. Third−party DSA providers may provide precomputed values for additional strength values.

It is possible to ask the key generator to use different values for P, Q, and G if the key pair generator supports the `DSAKeyPairGenerator` interface (`java.security.interfaces.DSAKeyPairGenerator`):

public interface DSAKeyPairGenerator
Provide a mechanism by which the DSA−specific parameters of the key pair engine can be manipulated.

There are two methods in this interface:

public void initialize(int modlen, boolean genParams, SecureRandom random)
Initialize the DSA key pair generator. The modulus length is the number of bits used to calculate the parameters; this must be any multiple of 64 between 512 and 1024. If `genParams` is `true`, then the P, Q, and G parameters will be generated for this new modulus length; otherwise, a precomputed value will be used (but precomputed values in the Sun security provider are available only for `modlen` values of 512 and 1024). If the modulus length is invalid, this method throws an `InvalidParameterException`.

public void initialize(DSAParams params, SecureRandom random)

Initialize the DSA key pair generator. The P, Q, and G parameters are set from the values passed in `params`. If the parameters are not correct, an `InvalidParameterException` is generated.

As with the `DSAKey` interface, a DSA key pair generator implements the `DSAKeyPairGenerator` interface for two purposes: for type identification, and to allow the programmer to initialize the key pair generator with the desired algorithm−specific parameters:

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
if (kpg instanceof DSAKeyPairGenerator) {
        DSAKeyPairGenerator dkpg = (DSAKeyPairGenerator) kpg;
        dkpg.initialize(512, true, new SecureRandom(  ));
}
else kpg.initialize(512);
```

In sum, this interface allows us to use the generic key pair generator interface while providing an escape clause that allows us to perform DSA−specific operations.

9.2.2 Implementing a Key Pair Generator

If you want to implement your own key pair generator −− either using a new algorithm or, more typically, a new implementation of a standard algorithm −− you create a subclass of the `KeyPairGenerator` class. The `KeyPairGenerator` class already extends the `KeyPairGeneratorSpi` class; this is one case where you don't extend the SPI class directly.

There are two abstract public methods of the key pair generator SPI that we must implement in our key pair generator: the `initialize( )` method and the `generateKeyPair( )` method. For this example, we'll generate a simple key pair that could be used for a simple rotation−based encryption scheme. In this scheme, the key serves as an offset that we add to each ASCII character −− hence, if the key is 1, an encryption based on this key converts the letter a to the letter b, and so on (the addition is performed with a modulus such that z will map to a). To support this encryption, then, we need to generate a public key that is simply a number between 1 and 25; the private key is simply the negative value of the public key.

We must also define a class to represent keys we're implementing.[1] We can do that with this class:

[1] This is true even if you're implementing an algorithm already implemented by the Sun security provider. The classes the Sun security provider uses to represent keys are not in the `java` package, so they are unavailable to us. So even if you're implementing DSA keys, you must still define classes that implement all the DSA interfaces we looked at earlier.

```
package javasec.samples.ch09;

import java.security.*;

public class XYZKey implements Key, PublicKey, PrivateKey {
    int rotValue;

    public String getAlgorithm(  ) {
        return "XYZ";
    }

    public String getFormat(  ) {
        return "XYZ Special Format";
    }

    public byte[] getEncoded(  ) {
        byte b[] = new byte[4];
        b[3] = (byte) ((rotValue << 24) & 0xff);
        b[2] = (byte) ((rotValue << 16) & 0xff);
        b[1] = (byte) ((rotValue <<  8) & 0xff);
        b[0] = (byte) ((rotValue <<  0) & 0xff);
        return b;
    }
}
```

The only data value our key class cares about is the value to be used as the index; for simplicity, we've made it a simple instance variable accessible only by classes in our package. Because this example is simple, we can use the same class as the interface for the public and the private key; normally, of course, public and private keys are not symmetric like this.

With this in place, we're ready to define our key pair generation class:

```
package javasec.samples.ch09;

import java.security.*;
import javasec.samples.ch08.*;

public class XYZKeyPairGenerator extends KeyPairGenerator {
    SecureRandom random;

    public XYZKeyPairGenerator(  ) {
        super("XYZ");
    }

    public void initialize(int strength, SecureRandom sr) {
        random = sr;
    }

    public KeyPair generateKeyPair(  ) {
        int rotValue = random.nextInt(  ) % 25;
        XYZKey pub = new XYZKey(  );
        XYZKey priv = new XYZKey(  );
        pub.rotValue = rotValue;
        priv.rotValue = -rotValue;
        KeyPair kp = new KeyPair(pub, priv);
        return kp;
    }

    public static void main(String[] args) throws Exception {
        Security.addProvider(new XYZProvider(  ));
        KeyPairGenerator kpg = KeyPairGenerator.getInstance("XYZ");
        kpg.initialize(0, new SecureRandom(  ));
        KeyPair kp = kpg.generateKeyPair(  );
        System.out.println("Got key pair " + kp);
    }
}
```

As a last step, we must install this class using the security provider that we examined in Chapter 8. Now obtaining a new key pair for the XYZ algorithm is as simple as substituting the string "XYZ" for the algorithm name when we request the key pair, as is shown in the `main( )` method.

9.2.3 The KeyGenerator Class

The `KeyGenerator` class (`javax.crypto.KeyGenerator`) is used to generate secret keys. This class is very similar to the `KeyPairGenerator` class except that it generates instances of secret keys instead of pairs of public and private keys:

public class KeyGenerator

Generate instances of secret keys for use by a symmetric encryption algorithm.

The `KeyGenerator` class is an engine within JCE. As such, it has all the hallmarks of a cryptographic engine. It has a complementary SPI and a set of public methods that are used to operate upon it, and its implementation must be registered with the security provider.

9.2.3.1 Using the KeyGenerator class

Like other engine classes, the `KeyGenerator` class doesn't have any public constructors. An instance of a `KeyGenerator` is obtained by calling one of these methods:

public static final KeyGenerator getInstance(String algorithm)
public static final KeyGenerator getInstance(String algorithm, String provider)

Return an object capable of generating secret keys that correspond to the given algorithm. These methods use the standard rules of searching the list of security providers in order to find an object that implements the desired algorithm. If the generator for the appropriate algorithm cannot be found, a `NoSuchAlgorithmException` is thrown; if the named provider cannot be found, a `NoSuchProviderException` is thrown.

JCE provides key generators that implement the following algorithms: Blowfish, DES, DESede, HmacMD5, and HmacSHA1. The first three algorithms are used in data encryption; the last two are used to calculate a message authentication code (MAC).

Once an object has been obtained with these methods, the generator must be initialized by calling one of these methods:

public final void init(SecureRandom sr)
public final void init(AlgorithmParameterSpec aps)
public final void init(AlgorithmParameterSpec aps, SecureRandom sr)
public final void init(int strength)
public final void init(int strength, SecureRandom sr)

Initialize the key generator. Like a key pair generator, the key generator needs a source of random numbers to generate its keys (in the second method, a default instance of the `SecureRandom` class will be used). In addition, some key generators can accept an algorithm parameter specification to initialize their keys (just as the key pair generator); however, for the DES−style keys generated by the SunJCE security provider, no algorithm parameter specification may be used.

A key generator does not have to be initialized explicitly, in which case it is initialized internally with a default instance of the `SecureRandom` class. However, it is up to the implementor of the engine class to make sure that this happens correctly; it is better to be sure your code will work by always initializing your key generator.

A secret key can be generated by calling this method:

public final SecretKey generateKey( )

Generate a secret key. A generator can produce multiple keys by repeatedly calling this method.

There are two additional methods in this class, both of which are informational:

public final String getAlgorithm( )

Return the string representing the name of the algorithm this generator supports.

public final Provider getProvider( )

Return the provider that was used to obtain this key generator.

In the next section, we'll show the very simple code needed to use this class to generate a secret key.

9.2.3.2 Implementing a KeyGenerator class

Implementing a key generator means creating a class that extends the `KeyGeneratorSpi` class (`javax.crypto.KeyGeneratorSpi`):

public abstract class KeyGeneratorSpi

Form the service provider interface class for the `KeyGenerator` class.

There are three protected methods of this class that we must implement if we want to provide an SPI for a key generator:

protected abstract SecretKey engineGenerateKey( )

Generate the secret key. This method should use the installed random number generator and (if applicable) the installed algorithm parameter specification to generate the secret key. If the engine has not been initialized, it is expected that this method will initialize the engine with a default instance of the `SecureRandom` class.

protected abstract void engineInit(SecureRandom sr)
protected abstract void engineInit(AlgorithmParameterSpec aps, SecureRandom sr)
public final void engineInit(int strength, SecureRandom sr)

Initialize the key generation engine with the given random number generator and, if applicable, algorithm parameter specification. If the class does not support initialization via an algorithm parameter specification, or if the specification is invalid, an `InvalidAlgorithmParameterException` is thrown.

Hence, a complete implementation might look like this. First, we must define a secret key type. Ours will hold the single integer used in XOR encryption:

```
package javasec.samples.ch09;

import javax.crypto.*;

public class XORKey implements SecretKey {
    int rotValue;

    XORKey(int value) {
        rotValue = value;
    }

    public String getAlgorithm(  ) {
        return "XOR";
    }

    public String getFormat(  ) {
        return "XOR Special Format";
    }

    public byte[] getEncoded(  ) {
        byte b[] = new byte[4];
        b[3] = (byte) ((rotValue << 24) & 0xff);
        b[2] = (byte) ((rotValue << 16) & 0xff);
        b[1] = (byte) ((rotValue <<  8) & 0xff);
        b[0] = (byte) ((rotValue <<  0) & 0xff);
        return b;
    }
}
```

Now we can define the key generator that creates these keys:

```
package javasec.samples.ch09;

import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javasec.samples.ch08.XYZProvider;

public class XORKeyGenerator extends KeyGeneratorSpi {
    SecureRandom sr;

    public XORKeyGenerator(  ) {
        XYZProvider.verifyForJCE(  );
    }

    public void engineInit(SecureRandom sr) {
        this.sr = sr;
    }

    public void engineInit(int len, SecureRandom sr) {
        if (len != 32)
            throw new IllegalArgumentException(
                                        "Only support 32 bit keys");
        this.sr = sr;
    }

    public void engineInit(AlgorithmParameterSpec aps, SecureRandom sr)
                          throws InvalidAlgorithmParameterException {
        throw new InvalidAlgorithmParameterException("Not supported");
    }

    public SecretKey engineGenerateKey(  ) {
        if (sr == null)
            sr = new SecureRandom(  );

        byte b[] = new byte[1];
        sr.nextBytes(b);
        return new XORKey(b[0]);
    }
}
```

Keys, of course, are usually longer than a single integer. However, unlike a public key/private key pair, there is not necessarily a mathematical requirement for generating a symmetric key. Such a requirement depends on the encryption algorithm the key will be used for, and some symmetric encryption algorithms require a key that is just an arbitrary sequence of bytes.

Remember that the key generator engine is a JCE engine. Because of this, we must verify it with our security provider, which is why the constructor invokes the `verifyForJCE( )` method of our provider.

9.3 Key Factories

There is another way to generate keys, and that is through the use of key factories. Key factories are most often used to import and export keys; the factory translates between an external format of the key that is easily transportable and the internal implementation of the key. There are certain types of keys, however, that can only be obtained via a key factory since there is no key generator to produce them (e.g., a password−based encryption key).

There are two external representations by which a key may be transmitted −− by its encoded format or by the parameters that were used to generate the key. Either of these representations may be encapsulated in a key specification, which is used to interact with the `KeyFactory` class (`java.security.KeyFactory`) and the `SecretKeyFactory` class (`javax.crypto.SecretKeyFactory`).

9.3.1 The KeyFactory Class

Converting asymmetric keys is done with this class:

public class KeyFactory
: Provide an infrastructure for importing and exporting keys according to the specific encoding format or parameters of the key.

9.3.1.1 Using the KeyFactory class

The `KeyFactory` class is an engine class, which provides the typical method of instantiating itself:

public static final KeyFactory getInstance(String alg)
public static final KeyFactory getInstance(String alg, String provider)
: Create a key factory capable of importing and exporting keys that were generated with the given algorithm. The class that implements the key factory comes from the named provider or is located according to the standard rules for provider engines. If a key factory that implements the given algorithm is not found, a `NoSuchAlgorithmException` is generated. If the named provider is not found, a `NoSuchProviderException` is generated.

A key factory presents the following public methods:

public final Provider getProvider( )
: Return the provider that implemented this particular key factory.

public final PublicKey generatePublic(KeySpec ks)
public final PrivateKey generatePrivate(KeySpec ks)
: These methods are used to import a key: they create the key based on the imported data that is held in the key specification object. If the key cannot be created, an `InvalidKeySpecException` is thrown.

public final KeySpec getKeySpec(Key key, Class keySpec)
: This method is used to export a key: it creates a key specification based on the actual key. If the key specification cannot be created, an `InvalidKeySpecException` is thrown.

public final Key translateKey(Key key)

Translate a key from an unknown source into a key that was generated from this object. This method can be used to convert the type of a key that was loaded from a different security provider (e.g., a DSA key generated from the XYZ provider -- type `com.xyz.DSAPrivateKey` -- could be converted to a DSA key generated from the Sun provider -- type `sun.security.provider.DSAPrivateKey`). If the key cannot be translated, an `InvalidKeyException` is generated.

public final String getAlgorithm( )

Return the algorithm this key factory supports.

We'll defer examples of these methods until we discuss the `KeySpec` class later.

9.3.1.2 Implementing a key factory

Like all engines, the key factory depends on a service provider interface class: the `KeyFactorySpi` class (`java.security.KeyFactorySpi`):

public abstract class KeyFactorySpi

Provide the set of methods necessary to implement a key factory that is capable of importing and exporting keys in a particular format.

The `KeyFactorySpi` class contains the following methods; since each of these methods is abstract, our class must provide an implementation of all of them:

protected abstract PublicKey engineGeneratePublic(KeySpec ks)
protected abstract PrivateKey engineGeneratePrivate(KeySpec ks)

Generate of the public or private key. Depending on the key specification, this means either decoding the data of the key or regenerating the key based on specific parameters to the key algorithm. If the key cannot be generated, an `InvalidKeyException` should be thrown.

protected abstract KeySpec engineGetKeySpec(Key key, Class keySpec)

Export the key. Depending on the key class specification, this means either encoding the data (e.g., by calling the `getEncoded( )` method) or saving the parameters that were used to generate the key. If the specification cannot be created, an `InvalidKeySpecException` should be thrown.

protected Key engineTranslateKey(Key key)

Perform the actual translation of the key. This is typically performed by translating the key to its specification and back. If the key cannot be translated, an `InvalidKeyException` should be thrown.

Although we show how to use a key factory later, we won't show how to implement one; the amount of code involved is large and relatively uninteresting. However, the online examples do contain a sample key factory implementation if you're interested in seeing one.

9.3.2 The SecretKeyFactory Class

The second engine that we'll look at is the `SecretKeyFactory` class (`javax.crypto.SecretKeyFactory`). Like the `KeyFactory` class, this class can convert from algorithmic or encoded key specifications to actual key objects and can translate key objects from one implementation to another. Unlike the `KeyFactory` class, which can only operate on public and private keys, the `SecretKeyFactory` class can operate only on secret keys:

public class SecretKeyFactory

Provide an engine that can translate between secret key specifications and secret key objects (and vice versa). This allows for secret keys to be imported and exported in a neutral format.

The interface to the `SecretKeyFactory` class is exactly the same at a conceptual level as the interface to the `KeyFactory`. At a programming level, this means that while most of the methods between the two classes have the same name and perform the same operation, they may require slightly different parameters: a secret key, rather than a public or private key. In addition, instead of methods to generate public or private keys, the `SecretKeyFactory` class contains this method:

public final SecretKey generateSecret(KeySpec ks)

Generate the secret key according to the given specification. If the specification is invalid, an `InvalidKeySpecException` is thrown.

The `SecretKeyFactory` class is an engine class; if you want, you may implement a secret key factory by subclassing the `SecretKeyFactorySpi` class (`javax.crypto.SecretKeyFactorySpi`). Because it is a JCE engine, the constructor of the secret key factory engine must invoke the `verifyForJCE( )` method of our sample provider (or execute similar code).

9.3.3 Key Specifications

Importing and exporting a key is based on classes that implement the `KeySpec` interface (`java.security.spec.KeySpec`):

public interface KeySpec

Identify a class as one that is able to hold data that can be used to generate a key.

The `KeySpec` interface is an empty interface; it is used for type identification only. This interface in turn forms the basis of algorithm−specific interfaces, each of which handles one method of importing a key.

9.3.3.1 The EncodedKeySpec class

Earlier we mentioned that the `Key` class must provide a `getEncoded( )` method for the key that outputs a series of bytes in a format specific to the type of key; this format is generally part of the specification for the key algorithm. For DSA keys, for example, the encoding format might be PKCS#8 or X.509. An encoded key specification holds the encoded data for a key and is defined by the `EncodedKeySpec` class (`java.security.spec.EncodedKeySpec`):

public abstract class EncodedKeySpec implements KeySpec

Provide an object to hold the encoded data of a key.

An encoded key specification can be operated on via these methods:

public abstract byte[] getEncoded( )

Return the actual encoded data held by the object. The array of bytes can be used later to instantiate a specific encoded key specification object.

public abstract String getFormat( )
: Return the string that represents the format of the encoded data (e.g., PKCS#8).

9.3.3.2 The AlgorithmParameterSpec interface

In addition to their encoded format, keys are typically able to be specified by providing the parameters to the algorithm that produced the key. Specifying keys in this manner is a function of the `AlgorithmParameterSpec` interface (`java.security.spec.AlgorithmParameterSpec`):

public interface AlgorithmParameterSpec
: Provide an infrastructure for specifying keys based on the parameters used to generate them.

Implementations of this interface have specific methods that are used to retrieve or set the parameters within the object.

9.3.4 A Key Factory Example

As we mentioned at the beginning of this section, the prime reason for key factories is that they give us the ability to import and export keys. Exporting a key specification is typically done by transmitting the individual data elements of the key specification (those individual elements vary by the type of key). Importing a key specification typically involves constructing the specification with the transmitted elements as parameters to the constructor.

Here's an example using a DSA algorithmic parameter specification. We'll look first at exporting a key:

```
package javasec.samples.ch09;

import java.security.*;
import java.security.spec.*;
import java.io.*;

public class Export {
    public static void main(String args[]) {
        try {
            KeyPairGenerator kpg = KeyPairGenerator.getInstance("DSA");
            kpg.initialize(512, new SecureRandom(  ));
            KeyPair kp = kpg.generateKeyPair(  );
            Class spec = Class.forName(
                            "java.security.spec.DSAPrivateKeySpec");
            KeyFactory kf = KeyFactory.getInstance("DSA");
            DSAPrivateKeySpec ks = (DSAPrivateKeySpec)
                                  kf.getKeySpec(kp.getPrivate(  ), spec);
            FileOutputStream fos = new FileOutputStream("exportedKey");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ks.getX(  ));
            oos.writeObject(ks.getP(  ));
            oos.writeObject(ks.getQ(  ));
            oos.writeObject(ks.getG(  ));
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

Two items are interesting in this code. First, one argument to the `getKeySpec( )` method is a class object, requiring us to construct the class object using the `forName( )` method (a somewhat unusual usage). Then, once we have the key specification itself, we have to figure out how to transmit the specification. Since in this case the specification is an algorithmic specification, we chose to write out the individual parameters from the specification.[2] If we had used an encoded key specification, we simply would have written out the byte array returned from the `getEncoded( )` method.

[2] The `DSAPrivateKeySpec` class −− like all key specification classes −− is not serializable itself. But for reasons that we'll discuss later, it's better not to serialize key classes that are to be imported into another Java virtual machine anyway.

We can import this key as follows:

```
package javasec.samples.ch09;

import java.security.*;
import java.security.spec.*;
import java.io.*;
import java.math.*;

public class Import {
    public static void main(String args[]) {
        try {
            FileInputStream fis = new FileInputStream("exportedKey");
            ObjectInputStream ois = new ObjectInputStream(fis);
            DSAPrivateKeySpec ks = new DSAPrivateKeySpec(
                        (BigInteger) ois.readObject(  ),
                        (BigInteger) ois.readObject(  ),
                        (BigInteger) ois.readObject(  ),
                        (BigInteger) ois.readObject(  ));
            KeyFactory kf = KeyFactory.getInstance("DSA");
            PrivateKey pk = kf.generatePrivate(ks);
            System.out.println("Got private key");
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

This example is predictably symmetric to exporting a key.

9.3.4.1 Existing key specification classes

Table 9−1 lists all the classes that can be used to import and export (or translate) keys. To use this table, find an appropriate key specification that you'd like to use. That's simply a matter of finding the specification that matches the type of key that you have (e.g., the specifications beginning with DH are for Diffie−Hellman keys). Next use the methods shown to export data from the key spec or to create a new key spec. Then you can use the key factory to create a key from the specification.

Note that the `SecretKeySpec` class is an exception to this last step: that class implements the `SecretKey` interface already. Once you've instantiated a `SecretKeySpec` object, you've created a secret key.

Table 9−1. Importing and Exporting Values from the Key Specification Classes

Key Specifications	Methods to Export Data	Methods to Import Data
`DESedeKeySpec` `(JCE)`	`byte[] getKey( )`	`DESedeKeySpec(byte[] buf)` `DESedeKeySpec(byte[] buf, int offset)`
`DESKeySpec` `(JCE)`	`byte[] getKey( )`	`DESKeySpec(byte[] buf)` `DESKeySpec(byte[] buf, int offset)`
`DHGenParameterSpec` `(JCE)`	`int getPrimeSize( )` `int getExponentSize( )`	`DHGenParameterSpec(int primeSize, int exponentSize)`
`DHParameterSpec` `(JCE)`	`BigInteger getP( )` `BigInteger getG( )` `int getL( )`	`DHParameterSpec(BigInteger p, BigInteger g)` `DHParameterSpec(BigInteger p, BigInteger g, int l)`
`DHPrivateKeySpec` `(JCE)`	`BigInteger getX( )` `BigInteger getP( )` `BigInteger getG( )` `int getL( )`	

```
DHPrivateKeySpec(
    BigInteger x,
    BigInteger p,
    BigInteger g)
DHPrivateKeySpec(
    BigInteger x,
    BigInteger p,
    BigInteger g, int l)

DHPublicKeySpec
(JCE)

BigInteger getY(  )
BigInteger getP(  )
BigInteger getG(  )
int getL(  )

DHPublicKeySpec(
    BigInteger x,
    BigInteger p,
    BigInteger g)
DHPublicKeySpec(
    BigInteger x,
    BigInteger p,
    BigInteger g, int l)

DSAParameterSpec
(core)

BigInteger getG(  )
BigInteger getP(  )
BigInteger getQ(  )

DSAParameterSpec(
    BigInteger g,
    BigInteger p,
    BigInteger q)

DSAPrivateKeySpec
(core)

BigInteger getG(  )
BigInteger getP(  )
BigInteger getQ(  )
BigInteger getX(  )

DSAPrivateKeySpec(
    BigInteger g,
    BigInteger p,
    BigInteger q,
    BigInteger x)

DSAPublicKeySpec
(core)

BigInteger getG(  )
BigInteger getP(  )
BigInteger getQ(  )
BigInteger getY(  )

DSAPublicKeySpec(
    BigInteger g,
    BigInteger p,
    BigInteger q,
    BigInteger y)

IvParameterSpec
(core)

byte[] getIV(  )

IvParameterSpec(
     byte[] buf)
IvParameterSpec(
     byte[] buf, int offset)

PBEKeySpec (JCE)

String getPassword(  )

PBEKeySpec(String pw)

PBEParameterSpec
(JCE)

int getIterationCount(  )
byte[] getSalt(  )

PBEParameterSpec(
    byte[] salt, int count)

PKCS8EncodedKey−Spec (core)

byte[] getEncoded(  )

PKCSEncodedKeySpec(
    byte[] key)

RC2ParameterSpec
(JCE)

byte[] getIV(  )
int getEffectiveKeyBits(  )

RC2ParameterSpec(
   int effective)
RC2ParameterSpec(
   int effective, byte[] iv)
RC2ParameterSpec(
   int effective, byte[] iv,
   int offset)

RC5ParameterSpec
(JCE)

byte[] getIV(  )
int getRounds(  )
int getVersion(  )
int getWordSize(  )

RC5ParameterSpec(
    int version, int rounds,
    int wordSize)
RC5ParameterSpec(
    int version, int rounds,
    int wordSize, byte[] iv)
RC5ParameterSpec(
    int version, int rounds,
    int wordSize, byte[] iv,
    int offset)

RSAKeyGenParameterSpec
(core)

int getKeysize(  )
BigInteger
   getPublicExponent(  )

RSAKeyGenParameterSpec(
    int size,
    BigInteger exp)

RSAPrivateKeySpec
(core)

BigInteger getModulus(  )
BigInteger
   getPrivateExponent(  )

RSAPrivateKeySpec(BigInteger mod, BigInteger exp)

RSAPrivateKeySpecCrt
(core)

BigInteger getModulus(  )
BigInteger
   getPrivateExponent(  )
BigInteger
   getPublicExponent(  )
BigInteger getPrimeP(  )
BigInteger getPrimeQ(  )
BigInteger
   getPrimeExponentP(  )
BigInteger
   getPrimeExponentQ(  )
BigInteger
   getCrtCoefficient(  )

RSAPrivateKeySpecCrt(
    BigInteger mod,
    BigInteger publicExp,
    BigInteger privExp,
    BigInteger primeP,
    BigInteger primeQ,
    BigInteger primeExpP,
    BigInteger primeExpQ,
    BigInteger crtCoeff)

RSAPublicKeySpec
(core)

BigInteger getModulus(  )
```

```
BigInteger
   getPublicExponent(  )

RSAPublicKeySpec(
    BigInteger mod,
    BigInteger exp)

SecretKeySpec
(JCE)

byte[] getEncoded(  )

SecretKeySpec(byte[] key,
    String Algorithm)
SecretKeySpec(byte[] key,
    int offset,
    String Algorithm)


X509EncodedKey−Spec
(core)

byte[] getEncoded(  )


X509EncodedKeySpec(
    byte[] key)
```

9.4 Certificates

When you are given a public and private key, you often need to provide other people with your public key. If you sign a digital document (using your private key), the recipient of that document will need your public key in order to verify your digital signature.

The inherent problem with a key is that it does not provide any information about the identity to which it belongs; a key is really just a sequence of seemingly arbitrary numbers. If I want you to accept a document that I digitally signed, I could mail you my public key, but you normally have no assurance that the key (and the original email) came from me at all. I could, of course, digitally sign the email so that you knew that it came from me, but there's a circular chain here −− without my public key, you cannot verify the digital signature. You would need my public key in order to authenticate the public key I've just sent you.

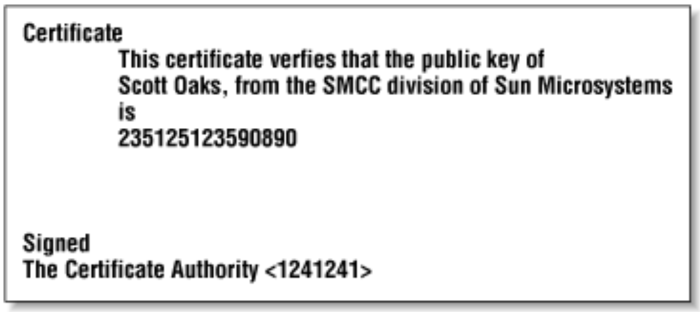
Certificates solve this problem by having a well−known entity (called a certificate authority, or CA) verify the public key that is being sent to you. A certificate can give you the assurance that the public key in the certificate does indeed belong to the entity that the certificate authority says it does. However, the certificate only validates the public key it contains: just because Fred sends you his public key in a valid certificate does not mean that Fred is to be trusted; it only means that the public key in question does in fact belong to Fred.

In practice, the key may not belong to Fred at all; certificate authorities have different levels at which they assess the identity of the entity named in the certificate. Some of these levels are very stringent and require the CA to do an extensive verification that Fred is who he says he is. Other levels are not stringent at all, and if Fred can produce a few dollars and a credit card, he is assumed to be Fred. Hence, one of the steps in the process of deciding whether or not to trust the entity named in the certificate includes the level at which the certificate authority generated the certificate. Each certificate authority varies in its approach to validating identities, and each publishes its approach to help you understand the potential risks involved in accepting such a certificate.

A certificate contains three pieces of information (as shown in Figure 9−2):

- The name of the entity for whom the certificate has been issued. This entity is referred to as the subject of the certificate.
- The public key associated with the subject.
- A digital signature that verifies the information in the certificate. The certificate is signed by the issuer of the certificate.

Figure 9−2. Logical representation of a certificate



Certificate
This certificate verfies that the public key of
Scott Oaks, from the SMCC division of Sun Microsystems
is
235125123590890

Signed
The Certificate Authority <1241241>

Because the certificate carries a digital signature of the certificate authority, we can verify that digital signature −− and if the verification succeeds, we can be assured that the public key in the certificate does in fact belong to the entity the certificate claims (subject to the level at which the CA verified the subject).

We still have a bootstrapping problem here −− how do we obtain the public key of the certificate authority? We could have a certificate that contains the public key of the certificate authority, but who is going to authenticate *that* certificate?

This bootstrapping problem is one reason why key management (see Chapter 10) is such a hard topic. Most Java implementations solve this problem by providing the public keys for certain well−known certificate authorities. This has worked well in practice, though it clearly is not an airtight solution (especially when the software is downloaded from some site on the Internet −− theoretically, the certificates that come with the software could be tampered with as they are in transit). Although there are various proposals to strengthen this model, for now we will assume that the certificate of at least one well−known certificate authority is delivered along with the Java application. This situation allows me to send you a certificate containing my public key; if the certificate is signed by a certificate authority you know about, you are assured that the public key actually belongs to me. Java 2, version 1.3 comes with 10 certificates of various authorities, which we'll discuss further in Chapter 10.

There are many well−known certificate authorities −− and therein lies another problem. I may send you a certificate that is signed by the United States Post Office, but that certificate authority may not be one of the certificate authorities you recognize. Simply sending a public key in a certificate does not mean that the recipient of the public key will accept it. A more important implication of this is that a key management system needs to be prepared to assign multiple certificates to a particular individual, potentially one from each of several certificate authorities.

Another implication of this profusion of certificate authorities is that certificates are often supplied as a chain. Let's say that you have the certificate of the U.S. Post Office certificate authority, and I want to send you my certificate that has been generated by the Acme Certificate company. In order for you to accept this certificate, I must send you a chain of certificates: my certificate (certified by the Acme Certificate company), and a certificate for the Acme Certificate company (certified by the U.S. Post Office). This chain of certificates may be arbitrarily long.

The last certificate in this chain −− that is, the public key for a certificate authority −− is stored in a certificate that is self−signed: the certificate authority has signed the certificate that contains its own public key. Self−signed certificates tend to crop up frequently in the Java world as well since the tools that come with the SDK will create self−signed certificates. The certificates are intended to be submitted to a certificate authority who will then return a CA−signed certificate. But there's no reason why the certificate itself can't be used as a valid certificate. Whether or not you want to accept a self−signed certificate is up to you, but it obviously carries certain risks.

Finally, for all this talk of certificates, you have to consider whether or not they are actually necessary to support your application. If you'll generally be receiving signed items from people you do not know (e.g., a signed jar file from a web site), then they are absolutely necessary. On the other hand, large−scale computer installations often consider using certificates to authenticate and validate their employees; this results in a computer system that has much better internal security than one that relies solely on passwords. However, the security does not stem from the certificate itself but from the use of public key cryptography. The computer installation can achieve the same level of security without using a certificate infrastructure.

Consider the security necessary to support XYZ Corporation's payroll application. When an employee wants to view her payroll statements, she must submit a digitally signed request to do so. Hence, XYZ should distribute to each employee a private key to be used to create the digital signature. XYZ can also store the employee's public keys in a database; when a request comes that claims to be from a particular employee, the payroll server can simply examine the database to obtain that employee's public key and verify the signature. No certificate is required in this case −− and in general, no certificate is required when the recipient of the digital signature is already known to have the public key of the entity that signed the data. For applications within a corporation, this is almost always the case.

We issue this caveat about certificates being necessary because certificate support in Java is not fully complete −− while it is possible to set up your own certificate authority to distribute the certificates for your company, it's very hard to write the necessary code to do that in Java. And, frankly, managing certificates is hard. Hence, we'll focus our discussion of the certificate API on accepting (i.e., validating) existing certificates.

Certificate: Class or Interface?

There's an unfortunate ambiguity in Java's use of the term "certificate." In Java 1.1, an interface called `java.security.Certificate` was introduced and used by the `javakey` utility and by the `appletviewer` when they used signed classes. The `Certificate` interface was implemented by platform−specific classes.

In Java 2, there is a new class called `java.security.cert.Certificate`. This class is the preferred class for all interactions with certificates and is used by the utilities provided with the Java 2 platform. The `java.security.Certificate` interface has been deprecated in Java 2.

In JSSE, there is yet another type of certificate: `javax.security.cert.Certificate`. These certificates are essentially equivalent to the standard Java 2 certificates; we'll explore them a little more in Chapter 14.

One problem where this manifests itself is with `import` statements. If you import the following packages:

```
import java.security.*;
import java.security.cert.*;
```

the compiler will be unable to reconcile the definition of `Certificate`. When dealing with certificates, you'll either need to refer to them by their fully qualified name or only import those classes in the security package that you explicitly need.

In the main text of this book, whenever we talk about a certificate object without qualifying its package name, we mean an instance of the `java.security.cert.Certificate` class (or one of its subclasses). Except for some examples in Appendix C, we will not show usage of the `Certificate` interface.

9.4.1 The Certificate Class

There are many formats that a certificate can take (depending on the cryptographic algorithms used to produce the certificate). Hence, the Java API abstracts the generic notion of a certificate with the `Certificate` class (`java.security.cert.Certificate`):

public abstract class Certificate

Provide the necessary (and very basic) operations to support a certificate.

Like many classes in the Java security package, the `Certificate` class is abstract; it relies upon application−specific classes to provide its implementation. In the case of the SDK, there are classes in the `sun` package that implement certain certificate formats (but more about that in just a bit).

There are three essential operations that you can perform upon a certificate:

public abstract byte[] getEncoded( )

Return a byte array of the certificate. All certificates must have a format in which they may be transmitted as a series of bytes, but the details of this encoding format are specific to the type of the certificate. If the encoding cannot be generated, a `CertificateEncodingException` is thrown.

public abstract void verify(PublicKey pk)
public abstract void verify(PublicKey pk, String provider)

Verify that the certificate is valid. In order to verify a certificate, you must have the public key of the certificate authority that issued it; a valid certificate is one in which the signature of the certificate authority is valid. A valid certificate does not imply anything about the trustworthiness of the certificate authority or the subject to which the certificate belongs; it merely means that the signature in the certificate is valid for the supplied public key. If the certificate is invalid, this method throws a `CertificateException`.

The signature is verified according to the digital signature details we'll examine in Chapter 12. The process of creating an object to verify the digital signature as well as the actual verification of the signature may throw a `NoSuchProviderException`, a `NoSuchAlgorithmException`, an `InvalidKeyException`, or a `SignatureException`.

public abstract PublicKey getPublicKey( )

Extract the public key from the certificate −− that is, the key that belongs to the subject the certificate vouches for.

These are the basic operations that are valid for any certificate. Notice that while we can encode a certificate into a byte array in order to transmit the certificate, there is nothing in the basic API that allows us to create a certificate from such a byte array. In fact, there's no practical way to instantiate a certificate object at all; the `Certificate` class is usually used as a base class from which individual certificate types are derived. Fortunately, the next class allows us to import certificates.

9.4.2 The CertificateFactory Class

If you need to import a certificate into a program, you do so by using the `CertificateFactory` class (`java.security.cert.CertificateFactory`). That class is an engine class, and it has the following interface:

public static CertificateFactory getInstance(String type)
public static CertificateFactory getInstance(String type, String provider)

Return a certificate factory that may be used to import certificates of the specified type (optionally implemented by the given provider). A `CertificateException` will be thrown if the given factory cannot be found or created; if the given provider is not found, a `NoSuchProviderException` will be thrown. The default Sun security provider has one certificate factory that works with certificates of type X509.

public String getProvider( )

Return the provider that implemented this factory.

public String getType( )

Return the type of certificates that this factory can import.

public final Certificate generateCertificate(InputStream is)

Return a certificate that has been read in from the specified input stream. For the default Sun security provider, the input stream must be an X509 certificate in RFC 1421 format (that is, a DER−encoded certificate that has been translated into 7−bit ASCII characters). Don't be worried by all those acronyms; this is the most common format for transmission of X509 certificates, and X509 certificates are the ones generated by most certificate authorities.

public final Collection generateCertificates(InputStream is)

Return a collection of certificates that have been defined in the given input stream. For the default Sun provider, the input stream in this case may have a single RFC 1421−formatted certificate, or it may contain a certificate chain in PKCS#7 format.

public final CRL generateCRL(InputStream is)

Define a certificate revocation list (CRL) from the data in the input stream.

public final Collection generateCRLs(InputStream is)

Define a collection of CRLs from the data in the input stream.

Note that the `CertificateFactory` class cannot generate a new certificate −− it may only import a certificate from an input stream. This is one reason why it's hard to provide a certificate authority based solely on the standard Java API. In the next section, we'll see an example of reading a certificate through this interface.

The `CertificateFactory` is an engine class, so it has a companion SPI class -- the `CertificateFactorySpi` class -- that can be used if you want to implement your own certificate factory. Implementing such a class follows the familiar rules of engine classes: you must define a constructor that takes the type name as a parameter and then, for each of the public methods listed above, you must implement a corresponding engine method with the same parameters. Certificates are complicated things, and parsing their encoding is a complicated procedure, so we won't bother showing an example of the engine class.

9.4.3 The X509Certificate Class

As we mentioned, there are many certificate formats that could be used by a key management system; one of the most common is the X509 format. X509 has gone through a few revisions; the version supported by the Java API is version 3. This format is an ANSI standard for certificates, and while there are PGP and other certificate formats in the world, the X509 format is dominant. This is the only format of certificate for which Java provides a standard API; if you want to support another certificate format, you must implement your own subclass of `Certificate`.

The `X509Certificate` class (`java.security.cert.X509Certificate`) is defined as follows:

public abstract class X509Certificate extends Certificate implements X509Extension

Provide an infrastructure to support X509 version 3 formatted certificates.

An X509 certificate has a number of properties that aren't shared by its base class:

- A start and end date: An X509 certificate is valid only for a certain period of time, as specified by these dates.
- A version: Various versions of the X509 standard exist; the default implementation of this class supports version 3 of the standard.
- A serial number: Each certificate that is issued by a certificate authority must have a unique serial number. The serial number is only unique for a particular authority so that the combination of serial number and certificate authority guarantee a unique certificate.
- The distinguished name[3] of the certificate authority.

  [3] See What's in a Name? in Chapter 10 for an explanation of distinguished names.
- The distinguished name of the subject represented by the certificate.

These properties can be retrieved with the following set of methods:

public abstract void checkValidity( )
public abstract void checkValidity(Date d)

Check that the specified date (or today if no date is specified) is within the start and end dates for which the certificate is valid. If the specified date is before the start date of the certificate, a `CertificateNotYetValidException` is thrown; if it is after the end date of the certificate, a `CertificateExpiredException` is thrown.

public abstract int getVersion( )

Return the version of the X509 specification that this certificate was created with. For the Sun implementation, this will be version 3.

public abstract BigInteger getSerialNumber( )

Return the serial number of the certificate.

public abstract Principal getIssuerDN( )

Extract the distinguished name of the certificate authority from the certificate and use that name to instantiate a principal object.

public abstract Principal getSubjectDN( )

Extract the distinguished name of the subject entity in the certificate and use that name to instantiate a principal object.

public abstract Date getNotBefore( )

Return the first date on which the certificate is valid.

public abstract Date getNotAfter( )

Return the date after which the certificate is invalid.

From a programmatic view, these are the most useful of the attributes of a certificate. If your X509 certificate is contained in the file *sdo.cer*, you could import and print out information about the certificate as follows:

```
package javasec.samples.ch09;

import java.security.cert.*;
import java.io.*;

public class PrintCert {
    public static void main(String args[]) {
        try {
            FileInputStream fr = new FileInputStream("sdo.cer");
            CertificateFactory cf =
                       CertificateFactory.getInstance("X509");
            X509Certificate c = (X509Certificate)
                                cf.generateCertificate(fr);
            System.out.println("Read in the following certificate:");
            System.out.println("\tCertificate for: " +
                                     c.getSubjectDN(  ));
            System.out.println("\tCertificate issued by: " +
                                     c.getIssuerDN(  ));
            System.out.println("\tThe certificate is valid from " +
                        c.getNotBefore() + " to " + c.getNotAfter(  ));
            System.out.println("\tCertificate SN# " +
                                     c.getSerialNumber(  ));
            System.out.println("\tGenerated with " +
                                     c.getSigAlgName(  ));
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

Running this program would produce the following output:

```
piccolo% java -classpath ../../.. javasec.samples.ch09.PrintCert
Read in the following certificate:
      Certificate for: CN=Scott Oaks, OU=SMCC, O=Sun, L=NY, ST=NY, C=US
      Certificate issued by: CN=Thawte Test CA Root, OU=TEST TEST TEST,
             O=Thawte Certification, ST=FOR TESTING PURPOSES ONLY, C=ZA
        The certificate is valid from Wed Apr 19 14:01:51 EDT 2000 to
             Sat May 20 14:01:51 EDT 2000
        Certificate SN# 6042116
        Generated with MD5withRSA
```

9.4.4 Advanced X509Certificate Methods

There are a number of other methods of the `X509Certificate` class. For the purposes of this book, these methods are not generally useful; they enable you to perform more introspection on the certificate itself. We'll list these methods here simply as a matter of record.

public abstract byte[] getTBSCertificate( )

Get the DER−encoded TBS certificate. The TBS certificate is the body of the actual certificate; it contains all the naming and key information held in the certificate. The only information in the actual certificate that is not held in the TBS certificate is the name of the algorithm used to sign the certificate and the signature itself.

The TBS certificate is used as the input data to the signature algorithm when the certificate is signed or verified.

public abstract byte[] getSignature( )

Get the raw signature bytes of the certificate. These bytes could be used to verify the signature explicitly (e.g., using the methods we'll describe in Chapter 12) instead of relying upon the `verify( )` method to do so.

public abstract String getSigAlgName( )

Return the name of the algorithm that was used to sign the certificate. For the Sun implementation, this will always be `SHA1withDSA`.

public String getSigAlgOID( )

Return the OID of the signature algorithm used to produce the certificate.

public abstract byte[] getSigAlgParams( )

Return the DER−encoded parameters that were used to generate the signature. In general, this will return `null` since the parameters are usually specified by the certificate authority's public key.

public abstract byte[] getIssuerUniqueID( )

Return the unique identifier for the issuer of the certificate. The presence of a unique identifier for each issuer allows the names to be reused, although in general it is recommended that certificates not make use of the unique identifier.

public abstract byte[] getSubjectUniqueID( )
: Return the unique identifier for the subject of the certificate (again, this is unused in general).

public abstract BitSet getKeyUsage( )
: Return the key usage extension, which defines the purpose of the key: the key may be used for digital signing, nonrepudiation, key encipherment, data encipherment, key agreement, certificate signing, and more. The key usage is an extension to the X509 specification and need not be present in all X509 certificates.

public abstract int getBasicConstraints( )
: An X509 certificate may contain an optional extension that identifies whether the subject of the certificate is a certificate authority. If the subject is a CA, this extension returns the number of certificates that may follow this certificate in a certification chain.

9.4.5 Revoked Certificates

Occasionally, a certificate authority needs to revoke a certificate it has issued −− perhaps the certificate was issued under false pretenses or maybe the user of the certificate has engaged in illegal conduct using the certificate. Under circumstances such as these, the expiration date attached to the certificate is insufficient protection; the certificate must be immediately invalidated.

This invalidation occurs as the result of a CRL −− a certificate revocation list. Certificate authorities are responsible for issuing certificate revocation lists that contain (predictably) a list of certificates the authority has revoked. Validators of certificates are required to consult this list before accepting the validity of a certificate.

Unfortunately, the means by which an authority issues a CRL is one of those areas that is in flux, and while the interfaces to support revoked certificates have been established, they are not completely integrated into most certificate systems. In particular, the `validate( )` method of the `Certificate` class doesn't automatically consult any CRL. The CRL itself is typically obtained in an out−of−band fashion (just as the certificates of the authority were obtained); once you have a CRL, you can check to see if a particular certificate in which you're interested is on the list.

While the notion of revoked certificates in not necessarily specific to an X509 certificate, the Java implementation is. Revoked certificates themselves are represented by the `X509CRLEntry` class (`java.security.cert.X509CRLEntry`):

public abstract class X509CRLEntry implements X509Extension
: The methods of this class are simple and are based upon the fields present in a revoked X509 certificate:

  public abstract BigInteger getSerialNumber( )
  : Return the serial number of the revoked certificate.

  public abstract Date getRevocationDate( )
  : Return the date on which the certificate was revoked.

  public abstract boolean hasExtensions( )
  : Indicate whether the implementation of the class has any X509 extensions.

Revoked certificates are modeled by the `X509CRL` class (`java.security.cert.X509CRL`):

public abstract class X509CRL implements X509Extension

Provide the support for an X509−based certificate revocation list.

Instances of the `X509CRLEntry` class are obtained by the `getInstance( )` method of the `CertificateFactory`. Once the class has been instantiated, you may operate upon it with these methods. As you can see, there is a strong synergy between the methods that are used to operate upon an X509 certificate and those used to operate upon a CRL:

public abstract void verify(PublicKey pk)
public abstract void verify(PublicKey pk, String sigProvider)

Verify that the signature that accompanied the CRL is valid. The public key should be the public key of the certificate authority that issued the CRL.

An error in the underlying signature object may generate a `NoSuchAlgorithmException`, a `NoSuchProviderException`, an `InvalidKeyException`, or a `SignatureException`.

public abstract int getVersion( )

Return the version of the CRL specification. The present version of the X509 CRL specification is 2.

public abstract Principal getIssuerDN( )

Extract the distinguished name of the issuer of the CRL and return a principal object that contains that name.

public abstract Date getThisUpdate( )

Extract and return the date when the authority issued this CRL.

public abstract Date getNextUpdate( )

Extract and return the date when the authority expects to issue its next CRL. This value may not be present in the CRL, in which case `null` is returned.

public abstract X509CRLEntry getRevokedCertificate(BigInteger bn)

Instantiate and return a revoked certificate object based on the given serial number. If the serial number is invalid, a `CRLException` is thrown.

public abstract Set getRevokedCertificates( )

Instantiate a revoked certificate object for each certificate in the CRL and return the set of those objects. This method may throw a `CRLException`.

public abstract byte[] getEncoded( )

Return the DER−encoded CRL itself. This method may throw a `CRLException`.

public abstract byte[] getTBSCertList( )

Return the DER−encoded TBS certificate list −− that is, all the data that came with the CRL aside from the name of the algorithm used to sign the CRL and the digital signature itself. This data can be used to verify the signature directly. Parsing of the underlying data may throw a `CRLException` or an `X509ExtensionException`.

public abstract byte[] getSignature( )

Return the actual bytes of the signature.

public abstract String getSigAlgName( )

Return the name of the signature algorithm that was used to sign the CRL.

public abstract String getSigAlgOID( )

Return the OID string of the signature algorithm that was used to sign the CRL.

public abstract byte[] getSigAlgParams( )

Return the DER−encoded algorithms used in the signature generation. This generally returns `null`, as those parameters (if any) usually accompany the authority's public key.

There is one more method of the `X509CRL` class, which it inherits from its superclass, the `CRL` class (`java.security.cert.CRL`):

public abstract boolean isRevoked(Certificate c)

Indicate whether or not the given certificate has been revoked by this CRL.

When all is said and done, the point of the `CRL` class (and the revoked certificate class) is to provide you with the tools necessary to see if a particular certificate has been invalidated. Your application should perform this checking; you might choose to implement it as follows:

```
package javasec.samples.ch09;

import java.security.*;
import java.security.cert.*;
import java.io.*;

public class TestCertificate {
    // Techniques to implement this method are shown
    // in the next chapter.
    PublicKey getPublicKey(Principal p) {
        return null;
    }

    // Implementations of this method depend on the CA in use and are
    // left to the reader.
    InputStream lookupCRLFile(Principal p) {
        return null;
    }

    public java.security.cert.Certificate
           importCertificate(byte data[]) throws CertificateException {
        X509Certificate c = null;
        try {
            CertificateFactory cf =
                                CertificateFactory.getInstance("X509");
            ByteArrayInputStream bais = new ByteArrayInputStream(data);
            c = (X509Certificate) cf.generateCertificate(bais);
            Principal p = c.getIssuerDN(  );
            PublicKey pk = getPublicKey(p);
            c.verify(pk);
            InputStream crlFile = lookupCRLFile(p);
            cf = CertificateFactory.getInstance("X509CRL");
            X509CRL crl = (X509CRL) cf.generateCRL(crlFile);
            if (crl.isRevoked(c))
                throw new CertificateException("Certificate revoked");
        } catch (NoSuchAlgorithmException nsae) {
            throw new CertificateException("Can't verify certificate");
        } catch (NoSuchProviderException nspe) {
            throw new CertificateException("Can't verify certificate");
        } catch (SignatureException se) {
            throw new CertificateException("Can't verify certificate");
        } catch (InvalidKeyException ike) {
            throw new CertificateException("Can't verify certificate");
        } catch (CRLException ce) {
            // treat as no crl
        }
        return c;
    }
}
```

This method encapsulates importing a certificate and checking its validity. It is passed the DER−encoded data of the certificate to check (this data must have been read from a file or other input stream, as we showed earlier). Then we consult the certificate to find out who issued it, obtain the public key of the issuer, and validate the certificate. Before we return, however, we obtain the latest CRL of the issuing authority and ensure that the certificate we're checking has not been revoked; if it has been, we throw a `CertificateException`.

We've glossed over two details in this method: how we obtain the public key of the authority that issued the certificate and how we get the CRL associated with that authority. Implementing these methods is the crux of a key/certificate management system, and we'll show some ideas on how to implement the key lookup in Chapter 10. Obtaining the CRL is slightly more problematic since you must have access to a source for the CRL data. Once you have that data, however, it's trivial to create the CRL via the `generateCRL( )` method.

9.5 Keys, Certificates, and Object Serialization

Before we conclude this chapter, a brief word on object serialization, keys, and certificates. Keys and certificates are often transmitted electronically, and a reasonable mechanism for transmitting them between Java programs is to send them as serialized objects. In theory −− and, most of the time, in practice −− this is a workable solution. If you modify some of the examples in this chapter to save and restore serialized keys or certificates, that will certainly work in a testing environment.

A problem arises, however, when you send these serialized objects between virtual machines that have two different security providers. Let's take the case of a DSA public key. When you create such a key with the Sun security provider, you get an instance of the `sun.security.provider.DSAPublicKey` class. When you create such a key with a third−party security provider, you may get an instance of the `com.xyz.XYZPublicKey` class. Although both public keys are extensions of the `PublicKey` class, they cannot be interchanged by object serialization. Serializing a public key created with the Sun security provider requires that the `sun.security.provider.DSAPublicKey` class be used, and deserialization creates an object of that type, no matter what security providers the deserializing virtual machine has installed. Whether or not the Sun security provider has been installed in the destination virtual machine is irrelevant. The process of deserializing the object uses that class if it is available, and deserialization fails if that class is not available.

Hence, while they are serializable objects, keys and certificates should only be transmitted as encoded data. For keys, you also have the option of transmitting the data contained in the key specification as we did earlier; the key specification classes are not serializable themselves, so you still have to rely on transmitting only the data that those objects contain.

This rule applies not only to keys and certificates that stand alone, but also to classes that embed one of those objects. Take, for example, this class:

```
public class Message implements Serializable {
        String msg;
        X509Certificate cert;
        byte signature[];
}
```

If you want to send an object of this class to a remote virtual machine (or save the object to a file), you should override the `writeObject( )` and `readObject( )` methods of the class so that when it is transmitted, the certificate is transmitted only as its encoded data and not as an instance of the `sun.security.x509.X509CertImpl` class. We'll do just that in Chapter 12.

9.6 Comparison with Previous Releases

Keys are present in Java 1.1 and the Java 2 platform, but much of the support for them that we've discussed here is only available in the Java 2 platform. Key factories and key specifications are only available in Java 2. In Java 1.1, you can get the encoded key data directly from a key, but that's a one−way operation; there's no practical way to import a key in Java 1.1.

In Java 1.1, the `KeyPairGenerator` class extends only the `Object` class; in Java 2, this class extends the `KeyPairGeneratorSpi` class. As is usual with this architecture, some of the methods we used are methods of the `KeyPairGenerator` class in Java 1.1 and methods of the `KeyPairGeneratorSpi` class in 1.2; for the developer, the end result is the same.

Java 1.1 only supports DSA keys. The RSA−based key interfaces were introduced in Java 2, version 1.2. However, there is no standard security provider in that release that implements RSA keys; only the security providers that come in Java 2, version 1.3 and with JSSE provide an RSA implementation.

9.7 Summary

Keys are a basic feature of any cryptographic system; they provide one of the inputs required to produce a digital signature (as well as other potential cryptographic operations). In this chapter, we looked at the basic classes that implement the notion of a key within the Java security package.

Keys are closely tied to the notion of certificates; a certificate contains a public key as well as an assurance from some known entity that the public key belongs to a specific entity. In a general sense, there are a great many things you can do with certificates, but for our purposes, we're interested in certificates only from the perspective of the certificate's user −− that is, we want to be able to import and verify a certificate, but we're not too interested in creating our own certificates or in becoming a certificate authority.

Given that the operations we want to perform on keys and certificates are simple −− importing and exporting those certificates −− you'd expect that we could leave our discussion of keys for the time being. Unfortunately, the topic of finding a key for a particular entity (which is really just a case of importing a key) is a particularly troublesome topic, which we'll examine in the next chapter.

Chapter 10. Key Management

In this chapter, we're going to discuss key management and the facilities in Java that enable key management. The problem of key management turns out to be a hard one to solve: there is no universally accepted approach to key management, and although many features in Java (and on the Internet) are available to assist with key management, all key management techniques remain very much works in progress.

Keys are important because they allow us to perform a number of cryptographic operations, from digital signatures to encrypted data streams. We'll discuss the details of these algorithms in the next few chapters. For now, it's enough to know that you must provide some sort of key or certificate for many of these algorithms: sometimes you need a private key, sometimes you need a secret key, and sometimes you need a public key contained within a certificate. The purpose of a key management system is to store such keys and allow you to retrieve them programatically (or through certain tools). A key management system may encompass other operations (it may, for example, provide information about the degree to which a particular individual should be trusted), but it exists primarily to serve up keys and certificates.

In this chapter, we'll discuss Java's key management system, which is built around the notion of a keystore. Keystores are created and manipulated though an administrative tool (`keytool`), and there is a Java API that allows you to use keystores programatically. We'll start this chapter by looking at `keytool`, which will allow us to become familiar with the concepts embodied by a keystore. Then we'll see how you can use the keystore programatically. Because Sun's implementation of the keystore is not necessarily suitable for all facilities (in particular, it is not the best system for enterprise−wide key management), we'll then look at the facilities available to build your own key management system. Finally, we'll conclude with other techniques to manage secret keys since secret keys are often managed outside of a traditional keystore.

10.1 Key Management Terms

There are a number of terms that are important in our discussion of Java's key management facilities:

keystore
: The keystore is the file that actually holds the set of keys and certificates. By convention, this file is called *.keystore* and is held in the user's home directory (*$HOME* on Unix systems, *C:\WINDOWS* on Microsoft Windows systems, and so on). However, there is great flexibility about where this file is located: the key management tools allow you to specify the location of the file, and the key management API allows you to use any arbitrary input stream. In fact, at the end of this chapter we'll discuss how the set of keys may be held in a persistent store like a centralized database.

alias
: Every key in the keystore belongs to an entity. An alias is a shortened, keystore−specific name for an entity that has a key or certificate in the keystore. I choose to store my public and private key in my local keystore under the alias "sdo"; if you have a copy of my public key certificate, you may use that alias, or you may use another alias (like "ScottOaks"). The alias used for a particular entity is completely up to the discretion of the individual who first enters that entity into the keystore.

DN (distinguished name)
: The distinguished name for an entity in the keystore is a subset of its full X.500 name. This is a long string; for example, my DN is:

```
CN=Scott Oaks, OU=JSD, O=Sun Microsystems, L=New York, S=NY, C=US
```

DNs are used by certificate authorities to refer to the entities to whom they supply a certificate. Hence, unlike an alias, the DN for a particular key is the same no matter what keystore it is located in: if I send you my public key, it will have the DN encoded in the public key's certificate.

However, nothing prevents me from having two public keys with different DNs (I might have one for personal use that omits references to my place of employment). And there is no guarantee that two unrelated individuals will not share the same DN (in fact, you can count on this type of namespace collision to occur).

The common name (CN) within a DN is usually the domain name of the organization to which the certificate belongs. In fact, SSL uses this convention to verify the identity of the server to which it connects.

What's in a Name?

X509 certificates (and many other ANSI standards) make use of the idea of a distinguished name (usually referred to as a DN). The distinguished name of an individual includes these fields:

Common name (CN)
The (full) common name of the individual.

Organizational unit (OU)
The unit the individual is associated with.

Organization (O)
The organization the individual is associated with.

Location (L)
The city where the individual is located.

State (S)
The state/province where the individual is located.

Country (C)
The country where the individual is located.

The DN specification allows other fields as well, although these are the only fields used internally in Java. The organization that is associated with an individual is typically the company the individual works for, but it can be any other organization (and of course, you may not be associated with an organization under a variety of circumstances).

The idea behind a DN is that it limits name duplication to some extent. There are other people named Scott Oaks in the world, but only one who has a DN of:

```
CN=Scott Oaks, OU=JSD, O=Sun Microsystems, L=NY, S=NY, C=US
```

On the other hand, this is not absolute; there are many nonunique DNs.

key entries

A keystore may hold two types of entries. The first type of entry is called a key entry. A key entry may hold either an asymmetric key pair (private key and public key certificate) or a single secret key. If the entry holds a key pair, it may store a chain of certificates: the first certificate always contains the public key of the entity. Other certificates may be present that establish a chain to the root certificate of the CA that issued the entity's certificate.

certificate entries

A certificate entry contains only a public key certificate; there is no private key associated with this entry. Certificate entries hold a single certificate rather than a chain, and the certificate is self−signed. These certificates are generally the root certificates of certificate authorities that you trust to issue certificates.

JKS, JCEKS, and PKCS12

The keystore is an engine within the Java API, and Sun's various security providers supply three different algorithms of the keystore. The default algorithm is JKS and is supplied by the security manager within the core API. It is capable of reading and storing key entries and certificate entries; however, the key entries can store only private keys. If you want to use the keystore for secret keys, you must use the JCEKS implementation, which is supplied by the security provider that comes with JCE. The JCEKS keystore can hold either private or secret keys for each key entry.

The private keys held by JKS or JCEKS are encrypted. The encryption used by JKS is weaker than that used by JCEKS; it was designed to pass the old export restrictions of the U.S. The JCEKS keystore provides a much stronger encryption.

For these two reasons, JCEKS is a preferable keystore; for compatibility reasons the default keystore remains JKS. Both the Java key management API and `keytool` allow you to specify the algorithm name when operating on a keystore, so you can use either algorithm at any time. However, if you want to change the default algorithm, you can edit the *$JREHOME/lib/security/java.security* file, find the entry for `keystore.type`, and change it to read:

```
keystore.type=JCEKS
```

The PKCS12 algorithm does not supply a fully−functional keystore. You can read a keystore in this format and export information (such as the encoded certificate) from that keystore, but you cannot write or modify a keystore in that format. This format is used to import certificates from your Netscape browser into your Java keystore, as we'll show a little later.

Trusted certificate authorities

Sun's implementation of Java comes with a set of trusted certificates from known certificate authorities. These certificates are held in *$JREHOME/lib/security/cacerts*, which is itself a keystore. That keystore holds ten certificate entries, each of which is the root certificate of a CA. This keystore should usually be considered read−only: you'll use it when you want to verify a certificate that was issued by one of these CAs, but you should not add your own key entries to this keystore.

If your enterprise has its own certificate authority, or if you obtain the root certificate from another well−known certificate authority, you may choose to install that certificate entry into the list of trusted certificate authorities. Otherwise, this keystore is best left unmodified. The CAs that are present in the *cacerts* file are listed here.

Thawte Personal Basic CA
Thawte Personal Freemail CA
Thawte Personal Premium CA
Thawte Premium Server CA
Thawte Server CA
Verisign Class 1 CA
Verisign Class 2 CA
Verisign Class 3 CA
Verisign Class 4 CA
Verisign Server CA
The password for this keystore is "changeit".

Note that these CAs are the all from the same company. Part of the difference between them is historical (Verisign and Thawte used to be different companies), but the more significant difference between them is the degree to which they verify the subject to whom the certificate is issued. You can get a certificate from the Thawte Personal Freemail CA for free and with relatively little information. If you want a certificate from the Verisign Server CA, you must provide a great deal of information (which Verisign will go to great lengths to verify), and you must pay for the certificate. As we mentioned in the last chapter, the level at which a particular CA issues a certificate is very important in deciding whether or not you should trust the holder of the certificate.

10.2 The keytool

At an administrative level, keys are managed by `keytool`, a utility supplied with the JRE. This tool allows you to create new keys, import digital certificates, export existing keys, and generally interact with the key management system.

The `keytool` has only a command−line interface; in this section, we'll look at the typical commands that add, modify, list, and delete entries in the keystore. Along the way, we'll see how you can create your own keys and certificates and how to get a valid certificate from an official certificate authority. As we understand the operations provided by `keytool`, we'll be poised to understand the underlying Java API that we'll examine later in this chapter.

10.2.1 Global Options to keytool

`Keytool` implements a number of global options −− options that are available to most of its commands. We'll list these as appropriate for each command, but here's an explanation of what they do:

−alias alias
Specify the alias the operation should apply to (e.g., `−alias sdo`). The default for this value is "mykey."

−dname distinguishedName
Specify the distinguished name. There is no default for this value, and if you do not specify it on the command line, you will be prompted to enter it when it is needed. Letting `keytool` prompt you is generally easier since the tool will prompt for the name one field at a time. Otherwise, you must enter

the entire name in one quoted string, like this:

```
-dname \
"CN=Scott Oaks, OU=JSD, O=Sun Microsystems, L=NY, S=NY, C=US"
```

−keypass password

Specify the password used to protect the entire keystore. Access to any element in the keystore requires this global password. If this password is not provided on the command line, you will be prompted for it. This is more secure than typing it on a command line or in a script where others might see it. Passwords must be at least six characters long.

For certain commands, the password may be omitted.

−keystore filename

Specify the name of the file that holds the keystore. The default value is *$HOME/.keystore*, as described before.

−storepass password

Specify the password used to protect a particular entry's private key. This is usually not (and should not be) the same as the global password. There should be a different password for each private key that is specific to that entry. This allows the keystore to be shared among many users. If the password is not provided on the command line, you will be prompted for it, which is the more secure way to enter this password.

−storetype storetype

Specify the type of keystore that the keytool should operate on. This defaults to the keystore type in the *java.security* file, which defaults to JKS, the keystore type provided by the Sun security provider.

−v

Verbose. Print some information about the operations `keytool` is performing.

Now we'll look at the various commands that are available within `keytool`, and along the way we'll build up a keystore that we'll use in examples in later chapters.

10.2.2 Creating a Key Entry

We'll start by creating a key entry that holds a private key and certificate. This is done by the following command, which creates the private key and a self−signed certificate that contains the corresponding public key:

−genkey

Generate a key pair and add that entry to the keystore. This command supports these global options:

−alias alias
−dname DN
−keypass keypass
−keystore keystore

−storepass storepass
−storetype storetype

It also supports these options:

−keyalg AlgorithmName
: Use the given algorithm to generate the key pair. The default for this option is DSA; you must use an algorithm name that is supported by a security provider that you have installed.

−keysize keysize
: Use the given keysize to initialize the key pair generator. The default value for this option is 1024; you must use a key size that is supported by the key algorithm you want to use.

−sigalg signatureAlgorithm
: Specify the signature algorithm that will be used to create the self−signed certificate; this defaults to `SHA1withDSA`, which is supported by the Sun security provider. If you've specified a different key algorithm (e.g., RSA), you'll have a different default signature algorithm (e.g., `SHA1withRSA`).

−validity nDays
: Specify the number of days for which the self−signed certificate should be valid. The default value for this option is 90 days.

The key entry that is created in this manner has the generated private key. In addition, the public key is placed into a self−signed certificate; that is, a certificate that identifies the holder of the public key (using the distinguished name argument) and is signed by the holder of the key itself. This is a valid certificate in all senses, although other sites will probably not accept the certificate since it was not issued by a known CA. However, the self−signed certificate can be used to obtain a certificate from a known CA, as we'll see in just a bit.

Here's how you use this command to create a key entry:

```
piccolo% keytool −genkey −alias sdo −keyalg RSA
Enter keystore password:  ******
What is your first and last name?
  [Unknown]:  Scott Oaks
What is the name of your organizational unit?
  [Unknown]:  JSD
What is the name of your organization?
  [Unknown]:  Sun Microsystems
What is the name of your City or Locality?
  [Unknown]:  NY
What is the name of your State or Province?
  [Unknown]:  NY
What is the two−letter country code for this unit?
  [Unknown]:  US
Is <CN=Scott Oaks, OU=JSD, O=Sun Microsystems, L=NY, S=NY, C=US> correct?
  [no]:  yes

Enter key password for <sdo>
        (RETURN if same as keystore password):  ******
```

At this point, we now have an entry for `sdo` in the keystore. That entry has a self−signed certificate; note that we had the tool prompt us for all the entries that comprise the DN rather than attempting to type it all in on the command line. We also chose to generate an RSA key pair since in later chapters, we'll want to use this key with SSL algorithms.

Be careful in selecting the values for the DN. Certain characters, such as commas, will become quoted. Unfortunately, quoted strings cannot be verified by other CAs. Hence, if you see quotes in the generated DN, reenter the information so that it is not quoted.

While it is possible for a keystore to hold a key entry that stores a secret key, `keytool` itself does not support creating such entries. Those entries can only be created programatically.

10.2.3 Generating a Certificate Request

If we want someone to accept the key we just generated, we need to obtain a certificate from a known CA. "Known" in this context means that we must already have the root certificate of the CA (e.g., in the *cacerts* file).

Which CA you choose is a complicated decision. As we've mentioned, CAs will take different steps to verify the identity of the person or organization to whom they issue a certificate. The certificate they issue will be verified by a different root certificate as a result: if you want a simple, free certificate with little verification of your identity, you can get a Thawte Personal Certificate. If you want something with more assurance as to your identity, you can pay for an SSL or developer's certificate from Thawte. That process takes longer since Thawte will do an extensive check to make sure that you are who you represent yourself to be.

So the issue here is to whom you will present your certificate and what level of verification they will accept. For our testing purposes, the Thawte Personal Certificate will do just fine; if you're presenting a certificate to a developer's association, they may require a Verisign Class 3 certificate, and so on. Of course, the converse of this relationship should also hold: when someone presents you with a certificate, you should check who issued it and what type it is in order to determine how careful the CA was in supplying the certificate.

In order to obtain a certificate from a CA, you must first generate a certificate signing request (CSR). The CSR contains the distinguished name and public key for a particular alias and is signed using the private key of the alias; the CA can then verify that signature and issue a certificate verifying the public key. CSRs are generated with this option:

−certreq

Generate a certificate signing request. This command supports the following global options:

−alias alias
−keypass keypass
−keystore keystore
−storepass storepass
−storetype storetype
−v

It also supports these options:

−sigalg signatureAlgorithm

Use the given algorithm to sign the CSR. The CSR must be signed by an algorithm the CA expects, and the algorithm must be consistent with the key being verified. The default algorithm will be based on the type of key held by the alias.

−file outputFile

Store the CSR in the given file. The format of the CSR is defined in PKCS#10. The default is to write the CSR to `System.out`.

Here's how to generate the CSR:

```
piccolo% keytool -certreq -alias sdo -file sdoCSR.cer
Enter keystore password:  ******
Enter key password for <sdo>: ******
```

If you used the same password for the keystore and the key itself, you are only prompted once for the password.

Once you have the CSR in a file, you must send it to the CA of your choice. Different CAs have different procedures for doing this, but all of them will send you back a certificate they have signed that verifies the public key you have sent to them. For simple testing, the quickest way to proceed is to register for a personal certificate at http://www.thawte.com/. Once you've received email from Thawte and continued with the registration process, you will arrive at https://www.thawte.com/cgi/personal/cert/enroll.exe; make sure to follow the section entitled "Developers of New Security Applications ONLY." Don't be dissuaded by the statements that you should only follow that link if you know what you're doing; that's the section on their web site that allows you to paste in a CSR. However, if you do this you must generate the initial keypair with a special value in the CN field; Thawte will tell you what that value is when you follow the links to request the certificate.

No matter which CA you use, you'll eventually be sent back the certificate, which will be in RFC 1421 format.

10.2.4 Importing a Certificate

When the response from the CA comes, we must save it to a file from which we can import it. In order to import the certificate, we must already have the root certificate in our list of trusted certificate authorities, or we must be prepared to accept the root certificate that `keytool` presents to us. To import the certificate, use this command:

−import

Import a certificate into the database. This command either creates a new certificate entry or imports a certificate for an existing key entry. This command supports the following global options:

−alias alias
−keypass keypass
−keystore keystore
−storepass storepass
−storetype storetype
−v

It also supports these options:

−file inputFile

The file containing the certificate that is being imported. The certificate must be in RFC 1421 format. The default is to read the data from `System.in`.

The certificate file sent by a CA will contain a certificate chain. The first certificate in the chain will be for the alias itself and will be issued by the certificate authority; the next certificate in the chain will be for the certificate authority and will be self−signed (a root certificate) or issued by another certificate authority, and so on until a self−signed certificate is present. While the encoding of the chain is defined by RFC 1421, the format of the chain itself is often referred to as a Netscape certificate chain or a PKCS #7 certificate chain. `Keytool` can read either format; if your CA gives you a choice of formats, pick either one.

−noprompt

Do not prompt the user about whether or not the certificate should be accepted. When this option is present, the certificate is always installed. Otherwise, if the root certificate in the chain is not from a trusted certificate authority, the user will be prompted whether or not to install the certificate chain.

When you import a certificate from an unrecognized CA, the information contained in that certificate is printed out; this information includes the fingerprint of the certificate and the distinguished names of the issuer and the principal. Well−known certificate authorities will publish their fingerprints (on the Web, in trade papers, and elsewhere). It is very important for you to verify the displayed fingerprint with the published fingerprint in order to verify that the certificate does indeed belong to the principal named in the certificate.

−trustcacerts

Use the *cacerts* file to obtain trusted certificates from certificate authorities that have signed the certificate that is being imported. Without this option, no CAs are considered trusted, and the user will always be asked whether or not to accept the certificate (unless, of course, the `noprompt` option is in effect).

If we saved the response from the CA in the file *sdo.cer*, here's how we'd import it into our keystore:

```
piccolo% keytool −import −file sdo.cer −alias sdo −trustcacerts
Enter keystore password:  ******
Certificate reply was installed in keystore
```

Assuming that the certificate is valid, this imports the new certificate into the keystore. The certificate is invalid if the public key for `sdo` does not match the previously defined public key in the database or if the certificate signature is invalid (which would be the case if data in the certificate had been modified in transit).

As a result of this command, the state of the `sdo` entry has significantly changed:

- When we created the key entry, the `sdo` entry had a single certificate; that certificate was issued by `sdo`.
- After the import command, the `sdo` entry has two or more certificates in its certificate chain: the first certificate is issued by the certificate authority and has a principal of `sdo`; the last certificate is the CA's self−signed certificate. There may be intermediate certificates in this chain.

10.2.5 Creating a Certificate Entry

Certificate entries in a keystore are always created by importing an existing certificate. The certificate may be the root certificate of a known CA (or the internal CA for your enterprise), or it may be a certificate that verifies the identity of someone with whom you will exchange information. For example, if I'm going to send you a digitally signed message, you must have my certificate (issued by a CA) within your keystore.

Certificate entries are created with the same import command that we just looked at. Let's say that I send you my certificate, and you've saved it to the file *fromsdo.cer*. You'd import it into your keystore like this:

```
piccolo% keytool −import −alias sdo −file fromsdo.cer

Enter keystore password:  ******
Owner: EmailAddress=scott.oaks@sun.com, CN=Thawte Freemail Member
Issuer: CN=Personal Freemail RSA 2000.8.30, OU=Certificate Services, O=Thawte,
L=Cape Town, ST=Western Cape, C=ZA
Serial number: 3df48
Valid from: Thu Dec 28 22:18:29 EST 2000 until: Fri Dec 28 22:18:29 EST 2001
Certificate fingerprints:
         MD5:  BE:E1:5C:54:E8:60:D4:09:7D:D8:C5:16:56:CA:72:5A
      SHA1: 4F:22:2D:E9:1C:7D:A6:D6:E4:1B:92:A5:CC:BE:DC:E8:DD:65:F6:45
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Note that in this example, we haven't used the *cacerts* file to verify the certificate automatically (and since the given CA doesn't exist in the *cacerts* file, that would fail anyway). This causes `keytool` to print out the certificate information; you should examine and verify its fingerprint before deciding whether or not to trust the certificate.

10.2.6 Modifying Keystore Entries

There is no way to modify a certificate entry in the keystore. You may delete an existing entry and add a new one if required.

There is one command that can modify the data within a key entry:

−selfcert

Change the certificate chain associated with the target key entry. Any previous certificates (including ones that may have been imported from a valid certificate authority) are deleted and replaced with a new self−signed certificate; this certificate can be used to generate a new CSR. The public and private keys associated with the alias are unchanged, but you may specify a new value for the DN on the command line. Hence, one use for this command is to change the DN for a particular entry before generating a CSR request.

This command supports the following global options:

−alias alias
−dname DN
−keypass keypass
−keystore keystore
−storepass storepass
−storetype storetype

It also supports these options:

−sigalg algorithmName

Use the given algorithm to generate the signature in the self−signed certificate.

−validity nDays
: The number of days for which the self−signed certificate is valid. The default is 90 days.

The *−keyclone* command is often used with this command, which can create a copy of the original entry before the DN is changed:

−keyclone
: Clone the target entry. The cloned entry will have the same private key and certificate chain as the original entry. This command supports the following global options:

  −alias alias
  −keypass keypass
  −keystore keystore
  −storepass storepass
  −storetype storetype
  −v

It also supports these options:

−dest newAlias
: The new alias name of the cloned entry. If this is not specified, you will be prompted for it.

−new newPassword
: The new password for the cloned entry. If this is not specified, you will be prompted for it. Again, it is more secure to respond to a prompt (because the password is masked) than it is to supply it in plain text at the command line.

To change the password associated with a particular key entry, use this command:

−keypasswd
: Change the password for the given key entry. This command supports the following global options:

  −alias alias
  −keystore keystore
  −storepass storePassword
  −storetype storetype
  −keypass originalPassword

It also supports this option:

−new newPassword
: Specify the new password for the entry. If this option is not supplied, you will be prompted for the new password.

Changing the password is one way to migrate entries from a JKS to a JCEKS keystore since you can specify a new storetype when you do so:

```
piccolo% keytool -keypasswd -alias sdo -storetype jceks
```

If you began with a JKS keystore, you'll end up with a JCEKS keystore after this command. Note that you can use this trick with other commands (e.g., the `storepasswd` command); anything that writes out a new keystore will change its format. However, the advantage of the JCEKS keystore is that the password associated with key entries is strongly encrypted, and the key entry password will not be reencrypted by other commands. Hence, to convert effectively between JKS and JCEKS, you must use the `keypasswd` command for each key alias in your keystore.

10.2.7 Deleting Keystore Entries

There is a single command to delete either a key entry or a certificate entry:

−delete

Delete the entry of the specified alias. If a certificate entry for a certificate authority is deleted, there is no effect upon key entries that have been validated by the authority. This command supports the following global options:

−alias alias
−keystore keystore
−storepass storepass
−storetype storetype
−v

10.2.8 Examining Keystore Data

If you want to examine one or more entries in the keystore, you may use the following commands:

−list

List (to `System.out`) one or more entries in the keystore. If an alias option is given to this command, only that alias will be listed; otherwise, all entries in the keystore are listed. You do not need to know the password for the keystore to use this command. This command supports the following global options:

−alias alias
−keystore keystore
−storepass storepass
−storetype storetype
−v

It also supports this option:

−rfc

When displaying certificates, display them in RFC 1421 standard. This option is incompatible with the *−v* option.

−export

Export the certificate for the given alias to a given file. The certificate is exported in RFC 1421 format. If the target alias is a certificate entry, that certificate is exported. Otherwise, the first certificate in the target key entry's certificate chain will be exported. If you need to send your certificate to another entity, you send it the file created by this command. This command supports the following global options:

−alias alias
−keystore keystore
−storepass storepass
−storetype storetype
−v

It also supports this option:

−file outputFile

The file in which to store the certificate. The default is to write the certificate to `System.out`.

−printcert

Print out a certificate. The input to this command must be a certificate in RFC 1421 format; this command will display that certificate in readable form so that you may verify its fingerprint. Unlike all other commands, this command does not use the keystore itself, and it requires no keystore passwords to operate. It supports the following global option:

−v

It also supports this option:

−file certificateFile

The file containing the RFC 1421 format certificate. The default is to read the certificate from `System.in`.

10.2.9 Miscellaneous Commands

There are two remaining commands to `keytool`. The first allows you to change the global password of the keystore:

−storepasswd

Change the global password of the keystore. This command supports the following global options:

−keystore keystore
−storepass storepass
−storetype storetype
−v

It also supports this option:

−new newPassword

The new global password for the keystore. If you do not specify this value, you will be prompted for it, which is more secure.

Finally, you can get a summary of all commands with this command:

−help
: Print out a summary of the usage of `keytool`.

10.2.10 Using Certificates from Netscape

If you have certificates that you've used in your Netscape browser, you can export them and use them with your Java programs as well. This is accomplished using the `PKCS12` keystore format. As we mentioned, this is presently a one−way operation: you can read a `PKCS12` keystore and export a certificate from it, but you cannot create or modify a `PKCS12` keystore.

The steps to accomplish this are as follows:

1. Export your certificate from Netscape.

   The exact details of this vary by Netscape release, but under the Tools menu, select Security Info (in Netscape 6, it's called Personal Security Manager). Select your certificate, and then export it. You can export it to any file; the normal extension to use for the file is *.p12*.
2. Read the keystore. If you exported the certificate to a file called *sdocer.p12*, this command will list the certificate:

   ```
   piccolo% keytool −list −keystore sdocer.p12 −storetype pkcs12
   Enter keystore password:  ******

   Keystore type: pkcs12
   Keystore provider: SunJSSE

   Your keystore contains 1 entry:

   scott oaks's verisign, inc. id, Sat Dec 30 18:39:54 EST 2000, keyEntry,
   Certificate fingerprint (MD5): 4D:09:08:11:95:FC:33:1C:6D:B1:15:2D:C3:FB:87:F8
   ```
3. Export and import the certificate, if desired. If you use the export command to export the certificate, you may import it into a JKS or JCEKS keystore. Besides integrating it into a single source, this allows you to make modifications to the entry, such as changing its password and alias.

10.3 The Key Management API

The `keytool` gives us the ability to create keys, obtain certificates, and so on. Now we'll turn our attention to using the key management facilities programatically: if you need to create a digital signature, you'll use the key management API to locate the correct key. Similarly, you may choose to store secret keys for data encryption in the keystore. The key management API allows us to read and write keystores and their entries.

In addition, the implementation of `keytool` has certain limitations: it cannot create entries that store secret keys, and it is difficult to share the keys in a `keytool` database among a widely dispersed group of people (like all the employees of XYZ Corporation). We can, however, use the key management API to create a key management system that has whatever features we require.

That framework is the ultimate goal of the following sections. First, however, let's take a look at the classes that make up the key management API. We begin with the notion of the identity to whom a key belongs. In Java's key management model, the association between a key and its owner is application−specific, but it is generally modeled on the `Principal` interface.

10.3.1 Principals

Classes that are concerned with identities and key management in the Java security package generally implement the `Principal` interface (`java.security.Principal`):

public interface Principal
: Provide an interface that supports the notion of an entity. In particular, principals have a name, but little else.

There is a single method that implementors of the `Principal` interface must implement:

public String getName( )
: Return the name of the principal. This is typically an X.500 distinguished name, but it may be any arbitrary name.

The only idea that the `Principal` interface abstracts is that principals have a name. The Java documentation states that a principal is anything that can have an identity, but don't be confused by that statement; the word "identity" is being overloaded in this context. There is an `Identity` class that implements the `Principal` interface, but there are classes implementing the `Principal` interface that are unrelated to the `Identity` class. In addition, although it is not officially deprecated, the `Identity` class is obsolete; it was used primarily in Java 1.1.

Further confusion about this interface can arise because there are two `Principal` types in Java 2: the `java.security.Principal` interface and the `org.omg.CORBA.Principal` class. These are unrelated, and we'll discuss only the `java.security.Principal` interface throughout this book.

The name that is stored in a principal is often an X.500 distinguished name (DN). That is particularly true when a principal is used in certain certificates (like X509 certificates); it is not an absolute requirement by any means.

There are other methods listed in the `Principal` interface −− namely, the `equals( )`, `toString( )`, and `hashCode( )` methods. There's no reason for those methods to be listed in the `Principal` interface since every class already inherits those methods from the `Object` class. If you implement the `Principal` interface, the only method you must implement is the `getName( )` method. You should make sure that the other methods of the `Principal` interface are implemented correctly −− but you should ensure these methods of the `Object` class are implemented correctly for all your classes, not just those that implement the `Principal` interface.

10.3.2 The KeyStore Class

The class that implements the keystore is the `KeyStore` class (`java.security.KeyStore`):

public class KeyStore
: Represent a set of private keys, aliases (entities), and their corresponding certificates. A keystore object is typically one that has been read in from disk; that is, the `KeyStore` object is an in−memory representation of the keystore file.

The `KeyStore` class is an engine class; there is a corresponding `KeyStoreSpi` class that you can use to write your own keystore (more about that a little later). As we've seen, the Sun−supplied algorithms for this engine are JKS, JCEKS, and PKCS12.

Instances of the `KeyStore` class are predictably obtained via this method:

public static final KeyStore getInstance(String type)
public static final KeyStore getInstance(String type, String provider)

Return an instance of the `KeyStore` class that implements the given algorithm, supplied by the given provider, if applicable.

If you do not want to hardwire the name of the keystore algorithm into your application, you may use this method to return the string that should be passed to the `getInstance( )` method:

public static final String getDefaultType( )

Return the default keystore algorithm for the environment. This value is obtained by looking for a property called `keystore.type` in the *java.security* file; Sun's version of Java sets the default value of this string to JKS.

When the keystore object is created, it is initially empty. Although the `getInstance( )` method has constructed the object, it is not expected that the object's constructor will read in a keystore from any particular location. The interaction between the keystore object and the keytool database comes via these two methods:

public final void load(InputStream is, char[] password)

Initialize the keystore from the data provided over the given input stream. The integrity of the keystore is protected by using a message digest: when the keystore is stored, a message digest that represents the data in the keystore is also stored. Before the digest is created, the password is added to the digest data; this means that the digest cannot be recreated from a keystore without knowledge of the password. This allows you to detect whether the keystore has been tampered with. The password for this method can be `null`, in which case the keystore is loaded and not verified.

It's somewhat misleading to call this parameter a password, although that's what the `javadoc` calls it, and that's the term used by `keytool`. If you pass `null` for the password, you'll always be able to read the keystore. Remember that a different password is used to decrypt the private keys in the keystore, so this isn't a security hole: if you don't have the password, you will be able to read only public certificates. If you use an incorrect password, an I/O exception is thrown.

You cannot require a password for the `load( )` method to succeed since the Sun implementation of the `Policy` class calls this method without a password when it constructs the information needed for the access controller. You may, of course, provide your own implementation of the `Policy` class that requires a password.

If the class required to support the underlying message digest is not available, a `NoSuchAlgorithmException` is thrown. An error in reading the data results in an `IOException`, and generic format errors in the data result in a `CertificateException`.

public final void store(OutputStream os, char[] password)

Store the keystore to the given output stream. The password is typically included in a digest calculation of the keystore; this digest is then written to the output stream as well (but again, your own implementation of this class could use the password differently). The format of the data is completely implementation dependent.

This method may throw an `IOException` if the output stream cannot be read, a `NoSuchAlgorithmException` if the class used to create the digest cannot be found, or a `CertificateException` if the keystore object contains a certificate that cannot be parsed.

There is no default file that holds the keystore. Within the core Java API, the only class that opens the keystore is the `PolicyFile` class, and that opens the keystore that is listed in the *java.policy* file(s). The tools that use the keystore (the `jarsigner` and `keytool` tools) allow you to use a command−line argument to specify the file that contains the keystore; they default to the file *.keystore* in the user's home directory. This is the convention your own programs will need to use. If your application needs to open the keystore (for example, to obtain a private key to sign an object), it should provide either a command−line argument or a property to specify the name of the file to open, and they should provide a reasonable default. Following convention, we'll use the *.keystore* file in the user's home directory in our examples.

As we've seen, a keystore is arranged in terms of alias names. Aliases are arbitrarily assigned to an entry; while the name embedded in the certificate for a particular entry may be a long, complicated, distinguished name, the alias for that entry can provide a shorter, easier−to−remember name. There are a number of simple methods in the `KeyStore` class that deal with these alias names:

public final Date getCreationDate(String alias)
Return the date on which the entry referenced by the given alias was created.

public final void deleteEntry(String alias)
Delete the entry referenced by the given alias from the keystore.

public final Enumeration aliases( )
Return an enumeration of all the aliases in the keystore.

public final boolean containsAlias(String alias)
Indicate whether the keystore contains an entry referenced by the given alias.

public final int size( )
Return the number of entries/aliases in the keystore.

public final boolean isKeyEntry(String alias)
public final boolean isCertificateEntry(String alias)
Indicate whether the given alias represents a key entry or a certificate entry.

public final Key getKey(String alias, char[] password)
Return the private or secret key for the entry associated with the given alias. For a certificate entry, this method returns `null`. An `UnrecoverableKeyException` is thrown if the key cannot be

retrieved (e.g., if the key has been damaged).

Retrieving a private key typically requires a password; this may or may not be the same password that was used to read the entire keystore. This allows private keys to be stored encrypted so they cannot be read without the appropriate password. If the class that provides encryption cannot be found, this method throws a `NoSuchAlgorithmException`.

public final Certificate[] getCertificateChain(String alias)

Return the certificate chain that verifies the entry associated with the given alias, which must represent a key entry. For an alias that represents a certificate entry, and for a key entry that stores a secret key, this method returns `null`.

public final Certificate getCertificate(String alias)

Return the certificate associated with the given alias. If the alias represents a key entry with a private key, the certificate returned is the user's certificate (that is, the first certificate in the entry's certificate chain); certificate entries have only a single certificate.

public final String getCertificateAlias(Certificate cert)

Return the alias that corresponds to the entry that matches the given certificate (using the `equals( )` method of certificate comparison). If no matches occur, `null` is returned.

public final void setKeyEntry(String alias, byte key[], Certificate chain[])
public final void setKeyEntry(String alias, Key k, char[] password, Certificate chain[])

Assign the given private or secret key and certificate chain to the key entry represented by the given alias, creating a new key entry if necessary. Any previous private key and certificate chain (or secret key) for this entry are lost; if the previous entry was a certificate entry, it now becomes a key entry. If the key is a secret key, the certificate chain should be `null`.

A `KeyStoreException` is thrown if the key entry cannot be encrypted by the internal encryption algorithm of the keystore. Note that when the key is passed in as a series of bytes, it is not encrypted -- in this case, you are expected to have performed the encryption yourself.

public final void setCertificateEntry(String alias, Certificate c)

Assign the given certificate to the certificate entry represented by the given alias, creating a new entry if necessary. If an entry for this alias already exists and is a key entry, a `KeyStoreException` is thrown. Otherwise, if an entry for this alias already exists, it is overwritten.

Note that there is no method that returns an entire entry; you must use the specific methods (such as the `getKey( )` method) to obtain the individual pieces of information you need.

These are the basic methods by which we can manage a keystore. We'll see examples of many of these methods throughout the rest of this book; for now, let's look at a simple example that handles basic operations on a keystore:

```
package javasec.samples.ch10;

import java.io.*;
import java.security.*;
import java.security.cert.*;

public class KeyStoreHandler {
    KeyStore ks;
    private char[] pw;

    // We'll use this to look up the keystore in the default location.
    // You can specify a password if you like, but this will also
    // work if you pass null (in which case the keystore isn't
    // verified).
    public KeyStoreHandler(char[] pw) {
        // Make a private copy so the original can be collected so
        // that other objects can't locate it.
        if (pw != null) {
            this.pw = new char[pw.length];
            System.arraycopy(pw, 0, this.pw, 0, pw.length);
        }
        else this.pw = null;
        // Load from the default location
        try {
            ks = KeyStore.getInstance(KeyStore.getDefaultType(  ));
            String fname = System.getProperty("user.home") +
                                    File.separator + ".keystore";
            FileInputStream fis = new FileInputStream(fname);
            ks.load(fis, pw);
        } catch (Exception e) {
            throw new IllegalArgumentException(e.toString(  ));
        }
    }

    public KeyStore getKeyStore(  ) {
        return ks;
    }

    // Store to the default location
    public void store(  ) throws FileNotFoundException,
                               KeyStoreException, IOException,
                               NoSuchAlgorithmException,
                               CertificateException {
        // If we didn't read with a password, we can't store
        if (pw == null) {
            throw new IllegalArgumentException("Can't store w/o pw");
        }
        FileOutputStream fos = new FileOutputStream(
                        System.getProperty("user.home") +
                        File.separator + ".keystore");
        ks.store(fos, pw);
        fos.close(  );
    }

    public static void main(String args[]) {
        try {
            KeyStore ks = new KeyStoreHandler(null).getKeyStore(  );
            if (ks.isKeyEntry(args[0])) {
                System.out.println(args[0] +
                                " is a key entry in the keystore");
                char c[] = new char[args[1].length(  )];
                args[1].getChars(0, c.length, c, 0);
                System.out.println("The private key for " + args[0] +
                            " is " + ks.getKey(args[0], c));
                java.security.cert.Certificate certs[] =
                                ks.getCertificateChain(args[0]);
                if (certs[0] instanceof X509Certificate) {
                    X509Certificate x509 = (X509Certificate) certs[0];
                    System.out.println(args[0] + " is really " +
                        x509.getSubjectDN(  ));
                }
                if (certs[certs.length - 1] instanceof
                                    X509Certificate) {
                    X509Certificate x509 = (X509Certificate)
                                        certs[certs.length - 1];
                    System.out.println(args[0] + " was verified by " +
                        x509.getIssuerDN(  ));
                }
            }
            else if (ks.isCertificateEntry(args[0])) {
                System.out.println(args[0] +
                            " is a certificate entry in the keystore");
                java.security.cert.Certificate c =
                                ks.getCertificate(args[0]);
                if (c instanceof X509Certificate) {
                    X509Certificate x509 = (X509Certificate) c;
                    System.out.println(args[0] + " is really " +
                        x509.getSubjectDN(  ));
                    System.out.println(args[0] + " was verified by " +
                        x509.getIssuerDN(  ));
                }
            }
            else {
                System.out.println(args[0] +
                        " is unknown to this keystore");
            }
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

We'll use this class in the rest of the book to manage the default keystore. It's `main( )` method (for testing) expects two arguments: the name of the entity in the keystore for which information is desired and the password that was used to encrypt the private key.

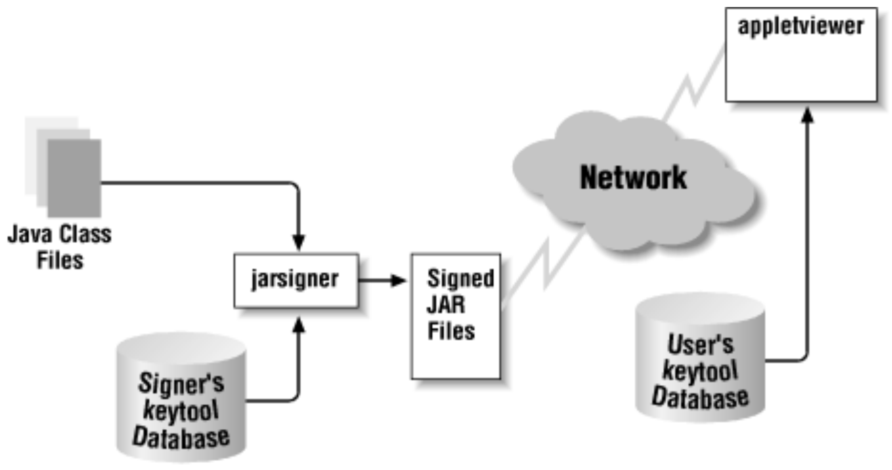
There are a number of points to pick out from this example. First, note that we constructed the keystore using the convention we mentioned earlier −− the *.keystore* file in the user's home directory.

After we've read in the data, the first thing we do is determine if the entry that we're interested in is a key entry or a certificate entry −− mostly so that we can handle the certificates for these entries differently. In the case of a key entry, we obtain the entire certificate chain and use the first entry in that chain to print out the DN for the entry while the last entry in the chain is used to print out the DN for the last certificate authority in the chain. For a certificate entry, our task is simpler: there is a single certificate, and we simply print out its information.

10.4 A Key Management Example

Now we'll proceed to a framework for enterprise−wide key management. Figure 10−1 shows the role of the keystore in the creation and execution of a signed jar file. The`jarsigner` utility consults the keystore for the private key of the entity that is signing the jar file. Once the signed jar file is produced, it is placed on a web server, where it can be downloaded into an `appletviewer` or the Java Plug−in. When the jar file is read on the remote system, the keystore is consulted in order to retrieve the public key of the entity that signed the jar file so that the jar file's signature can be verified.

Figure 10−1. the keytool database in a signed JAR file



Note that the two keystores in this example are (probably) separate files on separate machines. They probably have completely different entries as well −− even for the entry that represents the signer. The signer's entry in her own database must have the private key of the signer while the signer's entry in the user's database needs only a certificate for the signer. However, the keystore could (in this and all examples) be a shared database.

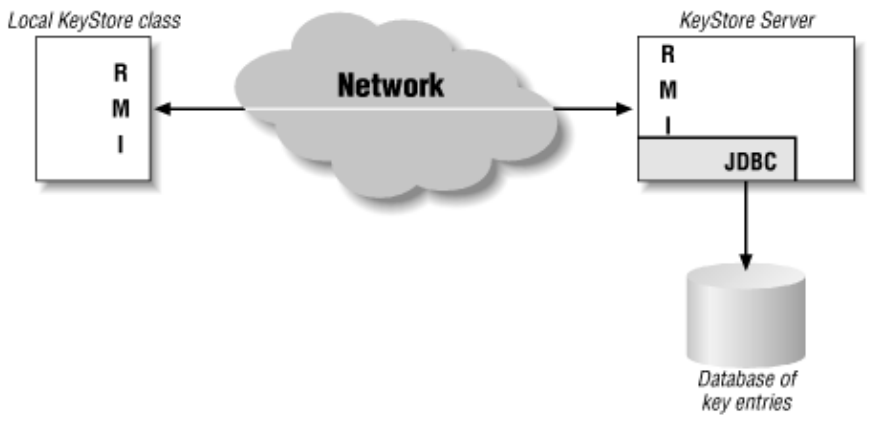
Since access to the private key of the signer is protected by a password, the signer and the end user are able to share a single database without concern that the end user may obtain access to the signer's private key (assuming that she keeps her password secret, of course). In the case of a corporate network, this flexibility is important since an enterprise may want to maintain a single database that contains the private keys of all of its employees as well as the certificates of all known external entities.

We could have these users share the keystore by using the appropriate filename in the application and the *java.policy* files. But sharing the keystore by a file is somewhat inefficient. If the global file is on a machine in New York and is referenced by a user in Tokyo, you will want to use a better network protocol to access it than a file−based protocol. In addition, the `load( )` method reads in the entire file. If there are 10,000 users in your corporate keystore database, you shouldn't need to read each entry into memory to find the one entry you are interested in using.

Hence, for many applications, you'll want to provide your own implementation of the `KeyStore` class. We'll show a very simple example here as a starting point for your own implementations. For the payroll application being deployed by XYZ Corporation, a database containing each employee in the corporation is necessary. The HR department could set up its own keystore for this purpose, but a similar keystore will be needed by the finance department to implement its 401K application; a better solution is to have a single keystore that is shared by all departments of XYZ Corporation.

In this case, the question becomes how best to share this keystore. A single global file would be too large for programs to read into memory and too unwieldy for administrators to distribute to all locations of XYZ Corporation. A better architecture is shown in Figure 10−2. Here, the application uses the security provider architecture to instantiate a new keystore object (of a class that we'll sketch out below). Unknown to the users of this object, the keystore class uses RMI (or CORBA, or any other distributed computing protocol) to talk to a remote server, which accesses the 10,000 employee records from a database set up for that purpose.

Figure 10−2. A distributed keystore example



Without getting bogged down in the details of the network and database programming required for this architecture, let's look at how the `KeyStore` class itself would be designed.

Implementing a keystore requires that we write a `KeyStoreSpi` class, just as with any engine class. For most methods in the `KeyStore` class, there is a corresponding abstract engine method in the `KeyStoreSpi` class that you must provide an implementation for. A complete list of these methods is given in Table 10−1.

Table 10−1. Engine Methods in the KeyStoreSpi Class

KeyStore Class	KeyStoreSpi class
`aliases`	`engineAliases`
`containsAlias`	`engineContainsAlias`
`deleteEntry`	`engineDeleteEntry`
`getCertificate`	`engineGetCertificate`
`getCertificateAlias`	`engineGetCertificateAlias`
`getCertificateChain`	`engineGetCertificateChain`

`getCreationDate`

`engineGetCreationDate`

`getKey`

`engineGetKey`

`isCertificateEntry`

`engineIsCertificateEntry`

`isKeyEntry`

`engineIsKeyEntry`

`load`

`engineLoad`

`setCertificateEntry`

`engineSetCertificateEntry`

`setKeyEntry`

`engineSetKeyEntry`

`size`

`engineSize`

`store`

`engineStore`

Many of the methods of our new class are simple passthroughs to the remote server. If the handle to the remote server is held in the instance variable `rks`, a typical method looks like this:

```
public Date engineGetCreationDate(String alias) {
    return rks.getCreationDate(alias);
}
```

The methods that could be implemented in this manner are:

```
engineGetKey(  )
engineGetCertificateChain(  )
engineGetCertificate(  )
engineGetCreationDate(  )
engineAliases(  )
engineContainsAlias(  )
engineSize(  )
engineIsKeyEntry(  )
engineIsCertificateEntry(  )
engineGetCertificateAlias(  )
```

On the other hand, many methods should probably throw an exception −− especially those methods that are designed to alter the keystore. In an architecture such as this one, changes to the keystore should probably be done through the database itself −− or at least through a different server than the server used by all employees in the corporation. Many functions may look simply like this:

```
public void engineSetKeyEntry(String alias, Key key,
                              char[] passphrase, Certificate chain[])
                                          throws KeyStoreException {
    throw new KeyStoreException("Can't change the keystore");
}
```

Methods that could be implemented in this manner are:

```
engineSetKeyEntry(  )
engineSetCertificateEntry(  )
engineDeleteEntry(  )
engineStore(  )
```

Note that we did not include the `engineLoad( )` method in the above list. The `engineLoad( )` method is useful to us because it allows the application to require a password from the user before a connection to the remote server can be made. This differs slightly from normal programming for this class. Typically, the `engineLoad( )` method is called with the input stream from which to read the keystore. In this case, the `engineLoad( )` method is expected to be called with a `null` input stream and sets up the connection to the remote server itself:

```
public void engineLoad(InputStream is, char[] password)
                       throws IOException, NoSuchAlgorithmException,
                                           CertificateException {
    RemoteKeyServer rks = (RemoteKeyServer)
               Naming.lookup("rmi://KSServer/DistributedKeyServer");
    if (!rks.authenticate(password)) {
        rks = null;
        throw new IOException("Incorrect password");
    }
}
```

Since the keystore database in this architecture cannot be written through the server, there is some question as to whether a password should be required to access the keystore at all (since there are individual passwords on the private keys). Every employee will potentially have access to the password (unless it is embedded into the application itself); you can decide if a password really adds security in that case. If no password is desired, the `engineLoad( )` method could be empty and the connection to the remote server could be made in the constructor.

On the server side, implementation of the required methods is simply a matter of making appropriate database calls:

```
public int engineSize(  ) {
    int sz = −1;
    try {
        Connection conn = connectToDatabase(  );
        Statement st = conn.createStatement(  );
        boolean restype = st.execute("select count(*) from entries");
        if (restype) {
            ResultSet rs = st.getResultSet(  );
            sz = Integer.parseInt(rs.getString(1));
        }
        st.cancel(  );
    } catch (Exception e) {
        ...
    }
    finally {
        return sz;
    }
}
```

This architecture works well because it allows the passwords for each of the private keys to be held in the database itself, so retrievals of private keys can easily test the password via a simple string comparison. Implementations of file−based keystores are more problematic: if the file is readable by the user, obviously the password cannot be stored in the file as a simple string. File−based keystores must store their passwords and their private keys in encrypted form, perhaps using the encryption APIs we'll examine in Chapter 13. Assuming that the database machine is secured, such encryption is not required in this architecture, unless you're concerned about people snooping the network looking for private keys; in that case, you should send the private keys as encrypted data to the user.

There are unlimited possibilities in the implementation of a keystore. One technique might be to create a floppy for each employee that contains only that employee's entry and to write a keystore class that looks for key entries from the file on the floppy and for certificate entries from a global file somewhere.[1] This type of implementation is very simple. The new keystore can contain two instances of the Sun `KeyStore` class that have read in both files, and it can use object delegation to implement all of its methods.

> [1] Of course, we don't want to use a floppy for this −− we want to use a Java−enabled smart card, although we don't all have smart card readers on our computers. At least, not yet...

Note that this type of two−tiered system is really the ideal. If the private keys are transmitted over the network, as in our previous case, then internal spies on the network might snoop the password used to retrieve the key or the private key that is sent back. If the private key is held locally, however, and only the public keys are retrieved from the remote key store, you have a cleaner implementation since the network data need not be encrypted.

10.4.1 Installing a KeyStore Class

In order to use an alternate keystore implementation, you must install your new class into a security provider. If necessary, you'll need to establish a convention by which the input stream that is opened for the `load( )` method is created −− unless your keystore does not require one at all (as, for example, our RMI−based keystore would not).

The `Policy` class uses the keystore in a predictable manner. Given these entries in the *java.policy* file:

```
keystore ${user.home}${/}.keystore
grant signedBy, "sdo", codeBase "http://piccolo/" {
        ...
}
```

the `Policy` class uses the keystore to look up the alias for `sdo`, retrieve `sdo`'s public key, and use that public key to verify any signature that comes from the site *piccolo*. It will open the file *$HOME/.keystore* and pass that input stream to the `load( )` method of the keystore class; if you're not using that file, you may simply ignore that input stream.

10.5 Secret Key Management

The key management techniques that we've discussed so far are ideally suited to managing asymmetric keys.

In fact, although we've mentioned that the keystore can hold secret keys, there are some problems with doing so.

When we discussed public and private key pairs, we talked about the bootstrapping issue involved with key distribution: the problem of obtaining the public key of a trusted certificate authority. In the case of key pairs, keeping the private key secret is of paramount importance. Anyone with access to the private key will be able to sign documents as the owner of the private key; he will also be able to decrypt data that is intended for the owner of the private key. Keeping the private key secret is made easier because both parties involved in the cryptographic transfer do not need to use it.

With symmetric keys, however, the bootstrapping issue is even harder to solve because both parties need access to the same key. The question then becomes how this key can be transmitted securely between the two parties in such a way that only those parties have access to the key.

One technique to do this is to use traditional (i.e., nonelectronic) means to distribute the key. The key could be put onto a floppy disk, for example, then mailed or otherwise distributed to the parties involved in the encryption. Or the key could be distributed in paper format, requiring the recipient of the key to type in the long string of hex digits.

Another technique is to use public key/private key encryption to encrypt the symmetric key and then send the encrypted key over the network. This allows the key to be sent electronically and used to set up the desired engine. This is a particularly attractive option for things like encryption because symmetric encryption is usually much faster than public key encryption. You can use the slower encryption for the secret key, then use the faster encryption for the rest of your data.

The final option is to use a key agreement algorithm. Key agreement algorithms exchange some public information between two parties so they each can calculate a shared secret key. However, they do not exchange enough information that eavesdroppers on the conversation can calculate the same shared key.

10.5.1 Secret Key Distribution

If you use traditional means to distribute a secret key, you have the problem of getting that key into your program. Your program can of course read the key directly from a file, but if many programs are going to share the secret key, you'd like to put that key into the keystore.

Unfortunately, `keytool` does not understand the idea of secret keys. If a keystore contains a secret key, then `keytool` will be able to list it, but there's no way to use `keytool` to introduce a secret key into the keystore.

Notice that in order for `keytool` to operate on secret keys, the keystore format must be JCEKS. You may specify that as the default format (as we showed earlier) in the *java.security* file, or you may specify that algorithm whenever you obtain the keystore.

To add a secret key entry into a keystore you have to use something like this program:

```
package javasec.samples.ch10;

import java.io.*;
import java.security.*;
import javax.crypto.*;

public class StoreKey {
    public static void main(String[] args) throws Exception {
        KeyStoreHandler ksh =
                new KeyStoreHandler(args[1].toCharArray(  ));
        KeyStore ks = ksh.getKeyStore(  );

        KeyGenerator kg = KeyGenerator.getInstance("DES");
        SecretKey sk = kg.generateKey(  );
        ks.setKeyEntry(args[0], sk, args[2].toCharArray(  ), null);
        ksh.store(  );
    }
}
```

You invoke this program with three arguments: first, the alias in which you'd like to store the newly−generated secret key; second, the global password for the keystore itself; and third, the password used specifically for this entry. This program generates the new secret key; optionally, you could read an encoded key or key specification from a file somewhere and build the secret key that way.

10.5.2 Secret Key Agreement

The other option for distributing secret keys is to use a key agreement algorithm, which is available only with the JCE security provider. In JCE, these algorithms are represented by the `KeyAgreement` class (`javax.crypto.KeyAgreement`):

public class KeyAgreement

Provide an engine for the implementation of a key agreement algorithm. This class allows for two cooperating parties to generate the same secret key while preventing parties unrelated to the agreement from generating the same key.

As an engine class, this class has no constructors, but it has the usual method to retrieve instances of the class:

public final KeyAgreement getInstance(String algorithm)
public final KeyAgreement getInstance(String algorithm, String provider)

Return an instance of the `KeyAgreement` class that implements the given algorithm, loaded either from the standard set of providers or from the named provider. If no suitable class that implements the algorithm can be found, a `NoSuchAlgorithmException` is generated; if the given provider cannot be found, a `NoSuchProviderException` is generated.

The interface to this class is very simple (much simpler than its use would indicate, as our example will show):

public final void init(Key k)
public final void init(Key k, SecureRandom sr)
public final void init(Key k, AlgorithmParameterSpec aps)
public final void init(Key k, AlgorithmParameterSpec aps, SecureRandom sr)

Initialize the key agreement engine. The parameter specifications (if present) will vary depending upon the underlying algorithm; if the parameters are invalid, of the incorrect class, or not supported, an `InvalidAlgorithmParameterException` is generated. This method will also perform the first phase of the key agreement protocol.

public final Key doPhase(Key key, boolean final)

Execute the next phase of the key agreement protocol. Key agreement protocols usually require a set of operations to be performed in a particular order. Each operation is represented in this class by a particular phase, which usually requires a key to succeed. If the provided key is not supported by the key agreement protocol, is incorrect for the current phase, or is otherwise invalid, an `InvalidKeyException` will be thrown.

The number of phases, along with the types of keys they require, vary drastically from key exchange algorithm to algorithm. Your security provider must document the types of keys required for each phase. In addition, you must specify which is the final phase of the protocol.

public final byte[] generateSecret( )
public final int generateSecret(byte[] secret, int offset)

Generate the bytes that represent the secret key; these bytes can then be used to create a `SecretKey` object. The type of that object will vary depending upon the algorithm implemented by this key agreement. The bytes are either returned from this argument or placed into the given array (starting at the given offset). In the latter case, if the array is not large enough to hold all the bytes a `ShortBufferException` is thrown. If all phases of the key agreement protocol have not been executed, an `IllegalStateException` is generated.

After this method has been called, the engine is reset and may be used to generate more secret keys (starting with a new call to the `init( )` method).

public final String getAlgorithm( )

Return the name of the algorithm implemented by this key agreement object.

public final Provider getProvider( )

Return the provider that implemented this key agreement.

Despite its simple interface, using the key agreement engine can be very complex. The `SunJCE` security provider implements one key agreement algorithm: Diffie−Hellman key agreement. This key agreement is based on the following protocol:

1. Alice (the first party in the exchange) generates a Diffie−Hellman public key/private key pair.
2. Alice transmits the public key and the algorithm specification of the key pair to Bob (the second party in the exchange).
3. Bob uses the algorithm specification to generate his own public and private keys; he sends the public key to Alice.
4. Alice uses her private key and Bob's public key to create a secret key. In the `KeyAgreement` class, this requires two phases: one that uses her private key and one that uses her public key.
5. Bob performs the same operations with his private key and Alice's public key. Due to the properties of a Diffie−Hellman key pair, this generates the same secret key Alice generated.

This secret key can then be used for a variety of operations: it can be used directly for creating MACs, or it can be converted into a DES key for use in a cipher. We'll see examples of those uses in later chapters.

Nothing in this key agreement protocol prevents someone from impersonating Bob −− Alice could exchange keys with me, I could say that I am Bob, and then Alice and I could exchange encrypted data. So even though the transmissions of the public keys do not need to be encrypted, they should be signed for maximum safety.

This algorithm works because of the properties of the Diffie−Hellman public key/private key pair. These keys are not suitable for use in an encryption algorithm; they are used only in a key agreement such as this.

Here's how a key agreement might be implemented:

```
package javasec.samples.ch10;

import java.math.*;
import java.security.*;
import java.security.spec.*;

import javax.crypto.*;
import javax.crypto.spec.*;
import javax.crypto.interfaces.*;

public class DHAgreement implements Runnable {
    byte bob[], alice[];
    boolean doneAlice = false;
    byte[] ciphertext;

    BigInteger aliceP, aliceG;
    int aliceL;

    public synchronized void run(  ) {
        if (!doneAlice) {
            doneAlice = true;
            doAlice(  );
        }
        else doBob(  );
    }

    public synchronized void doAlice(  ) {
        try {
            // Step 1:  Alice generates a key pair
            KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
            kpg.initialize(1024);
            KeyPair kp = kpg.generateKeyPair(  );

            // Step 2:  Alice sends the public key and the
            //         Diffie-Hellman key parameters to Bob
            Class dhClass = Class.forName(
                                "javax.crypto.spec.DHParameterSpec");
            DHParameterSpec dhSpec = (
                            (DHPublicKey) kp.getPublic()).getParams(  );
            aliceG = dhSpec.getG(  );
            aliceP = dhSpec.getP(  );
            aliceL = dhSpec.getL(  );
            alice = kp.getPublic().getEncoded(  );
            notify(  );

            // Step 4 part 1:  Alice performs the first phase of the
            //        protocol with her private key
            KeyAgreement ka = KeyAgreement.getInstance("DH");
            ka.init(kp.getPrivate(  ));

            // Step 4 part 2:  Alice performs the second phase of the
            //        protocol with Bob's public key
            while (bob == null) {
                wait(  );
            }
            KeyFactory kf = KeyFactory.getInstance("DH");
            X509EncodedKeySpec x509Spec = new X509EncodedKeySpec(bob);
            PublicKey pk = kf.generatePublic(x509Spec);
            ka.doPhase(pk, true);

            // Step 4 part 3:  Alice can generate the secret key
            byte secret[] = ka.generateSecret(  );

            // Step 6:  Alice converts a secret key
            SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
            DESKeySpec desSpec = new DESKeySpec(secret);
            SecretKey key = skf.generateSecret(desSpec);

            // Step 7:  Alice encrypts data with the key and sends
            //         the encrypted data to Bob
            Cipher c = Cipher.getInstance("DES/ECB/PKCS5Padding");
            c.init(Cipher.ENCRYPT_MODE, key);
            ciphertext = c.doFinal(
                        "Stand and unfold yourself".getBytes(  ));
            notify(  );
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }

    public synchronized void doBob(  ) {
        try {
            // Step 3:  Bob uses the parameters supplied by Alice
            //         to generate a key pair and sends the public key
            while (alice == null) {
                wait(  );
            }
            KeyPairGenerator kpg = KeyPairGenerator.getInstance("DH");
            DHParameterSpec dhSpec = new DHParameterSpec(
                                aliceP, aliceG, aliceL);
            kpg.initialize(dhSpec);
            KeyPair kp = kpg.generateKeyPair(  );
            bob = kp.getPublic().getEncoded(  );
            notify(  );

            // Step 5 part 1:  Bob uses his private key to perform the
            //         first phase of the protocol
            KeyAgreement ka = KeyAgreement.getInstance("DH");
            ka.init(kp.getPrivate(  ));

            // Step 5 part 2:  Bob uses Alice's public key to perform
            //         the second phase of the protocol.
            KeyFactory kf = KeyFactory.getInstance("DH");
            X509EncodedKeySpec x509Spec =
                            new X509EncodedKeySpec(alice);
            PublicKey pk = kf.generatePublic(x509Spec);
            ka.doPhase(pk, true);

            // Step 5 part 3:  Bob generates the secret key
            byte secret[] = ka.generateSecret(  );

            // Step 6:  Bob generates a DES key
            SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
            DESKeySpec desSpec = new DESKeySpec(secret);
            SecretKey key = skf.generateSecret(desSpec);

            // Step 8:  Bob receives the encrypted text and decrypts it
            Cipher c = Cipher.getInstance("DES/ECB/PKCS5Padding");
            c.init(Cipher.DECRYPT_MODE, key);
            while (ciphertext == null) {
                wait(  );
            }
            byte plaintext[] = c.doFinal(ciphertext);
            System.out.println("Bob got the string " +
                        new String(plaintext));
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }

    public static void main(String args[]) {
        DHAgreement test = new DHAgreement(  );
        new Thread(test).start(  );            // Starts Alice
        new Thread(test).start(  );            // Starts Bob
    }
}
```

Note that this example uses the `Cipher` class; see Chapter 13 for more details about that class.

In typical usage, of course, Bob and Alice would be executing code in different classes, probably on different machines. We've shown the code here using two threads in a shared object so that you can run the example more easily (although beware: generating a Diffie−Hellman key is an expensive operation, especially for a size of 1024; a size of 512 will be better for testing). Our second reason for showing the example like this is to make explicit the points at which the protocol must be synchronized: Alice must wait for certain information from Bob, Bob must wait for certain information from Alice, and both must perform the operations in the order specified. Once the secret key has been created, however, they may send and receive encrypted data at will.

Otherwise, despite its length, this example merely uses a lot of the techniques we've been talking about in the past two chapters. Keys are generated, they are transmitted in neutral (encoded) format, they are reformed by their recipient, and both sides can continue.

The `KeyAgreement` class is an engine class, and you can create your own implementations of it by subclassing the `KeyAgreementSpi` class (`javax.crypto.KeyAgreementSpi`). Remember that this is a JCE engine class, so you must perform the appropriate steps in the constructor of your engine to verify the JCE installation.

10.6 Comparison with Previous Releases

The fluidity of key management is evident in the progress of Java itself. Key management with the 1.1 API is very different from key management in Java 2. Further complicating this picture is the fact that no Java−enabled browser (except the Java 2 Plug−In) uses the technique for key management that comes with the JRE. Each requires keys to be kept in a different key database, and each uses a different technique to store and retrieve keys from that application−specific database.

As a developer, that means you must adopt different key management features depending on your target platform. If your target platform is Java 2 applications and Java 2 applets run through the Java Plug−in, then you can use this key management facility. If you must support applets run in Internet Explorer or versions of Netscape Navigator before Netscape 6, then you must use Microsoft− or Netscape−specific key management techniques. And if you're targeting Java 1.1 applications, you must use Java 1.1 facilities.

There are no keystores in Java 1.1. If you must implement a key management system under Java 1.1, you'll need to use the `IdentityScope` class. The `IdentityScope` class has been deprecated in Java 2.

Java 1.1 comes with a key management system that is based upon the `javakey` utility. `javakey` has several limitations; in particular, it stores public and private keys in the same, unprotected location (often called an identity database). This allows anyone with access to the `javakey` database to determine all the keys that were stored in the file. Since access is required to obtain your own private key to generate your own digital signature, this essentially gives all users access to each other's keys. This problem was a limitation of the `javakey` utility itself. It's possible to use the 1.1 classes to write a key database in such a way that your private key is held separately from a group of public keys (see Appendix C).

Although the keystore in Java 2 is incompatible with the identity database in 1.1, `keytool` is capable of converting between the two. To convert a 1.1 identity database to a Java 2 keystore, use this command:

−identitydb
: Convert a 1.1 identity database. This command has the following global options:

    −v
    −keystore keystore
    −keypass keypass
    −storepass storepass
    −storetype storetype

It also supports this option:

−file db_file
: The filename of the 1.1 identity database. The default for this is *identitydb.obj* in the user's home directory.

With this command, each trusted entry in the identity database will be created as a key entry in the keystore. All other entries in the identity database will be ignored.

10.7 Summary

In this chapter we examined the key management facilities of Java. Key management revolves around keys and certificates −− ideas we've already discussed −− but it also depends upon the notion of an identity −− an individual or a corporation −− and the idea that a particular identity can be certified.

Key management in Java can be handled either programmatically with the standard Java API or with the key management tool `keytool`. `keytool` itself is a good example of how the programming API can be used, although there are some trade−offs involved here; for example, loading a large keystore is not necessarily the most appropriate choice for a thin−client application. Fortunately, the security package gives us the necessary tools to implement our own keystore when that is appropriate.

For all the time we've spent on them, keys are not interesting by themselves. They are interesting for what they allow us to do, which among other things includes the ability to operate on a digital signature. In the next chapters, we'll look at message digests and digital signatures, their relationship to keys, and the operations that all this enables us to perform.

Chapter 11. Message Digests

In this chapter, we're going to look at the API that implements the ability to create and verify message digests. The ability to create a message digest is one of the standard engines provided by the Sun default security provider, and there are engines that manipulate digests in the Java Cryptography Extension as well. You can therefore reasonably expect every Java implementation to create message digests.

Message digests are the simplest of the standard engines that compose the security provider architecture. They provide the first link in creating and verifying a digital signature `--` one of the most important goals of the provider architecture. However, message digests are useful entities in their own right since a message digest can verify that data has not been tampered with `--` up to a point. As we'll see, there are certain limitations on the security of a message digest that is transmitted along with the data it represents.

We'll examine how developers can use the message digest in this chapter and also explore how a security provider can implement her own message digest algorithm.

11.1 Using the Message Digest Class

Message digests are implemented using the `MessageDigest` class (`java.security.MessageDigest`):

public abstract class MessageDigest extends MessageDigestSpi

Implement operations to create and verify a message digest.

Like all engine classes, instances of the message digest are obtained through one of these methods:

public static MessageDigest getInstance(String algorithm)
public static MessageDigest getInstance(String algorithm, String provider)

Return an instance of the message digest class that implements the given algorithm, optionally using the given provider. If no provider can be found that implements the given algorithm, a `NoSuchAlgorithmException` is thrown. If the named provider can't be found, a `NoSuchProviderException` is thrown.

Once a message digest object has been obtained, the developer can operate on that object with these methods:

public void update(byte input)
public void update(byte[] input)
public void update(byte[] input, int offset, int length)

Add the specified data to the digest. The first of these methods adds a single byte to the data, the second adds the entire array of bytes, and the third adds only the specified subset of the array of data.

These methods may be called in any order and any number of times to add the desired data to the digest. Consecutive calls to these methods append data to the internal accumulation of data over which the digest will be calculated.

public byte[] digest( )
public byte[] digest(byte[] input)

Compute the message digest on the accumulated data (optionally adding the specified data before performing the computation). The resulting digest is returned as a byte array. Once a digest has been calculated, the internal state of the algorithm is reset so that the object may be reused at this point to create a new message digest.

public int digest(byte[] output, int offset, int len)

Compute the message digest on the accumulated data and place the answer into the provided array, starting at the given offset and copying at most `len` bytes. Most implementations do not return a partial digest, so if the amount of space in the buffer (taking into account its offset) is not sufficient to store the digest, a `DigestException` is thrown. This method returns the size of the digest.

public static boolean isEqual(byte digestA[], byte digestB[])

Compare two digests for equality. Two digests are considered equal only if each byte in the first digest is exactly equal to each byte in the second digest and the digests are the same length.

public void reset( )

Reset the digest object by discarding all accumulated data and resetting the algorithm that is used to implement the digest. This is equivalent to creating a new instance of the object. In addition, this method throws away any information that the `toString( )` method would have printed (described later in this list).

public final String getAlgorithm( )

Return the string representing the algorithm name (e.g., SHA).

public String toString( )

A string representation of a digest by default contains the name of the class implementing the digest, the words "Message Digest," and the bytes that were returned by a previous call to the `digest( )` method. If the `digest( )` method has not been called, or if the `reset( )` method has been called, then instead of the digest, "<incomplete>" is printed. An example string looks like:

```
sun.security.provider.SHA Message Digest \
                <0a808982fee54fd74a86aae72eff7991328ff32b>
```

public Object clone( ) throws CloneNotSupportedException

Return a clone of the object. Message digest implementations need to implement the `clone( )` method because some internal operations on the digest object require a call to the `digest( )` method, which resets the digest. These operations are typically done on a clone of the object so that the state of the original object is not changed.

public final int getDigestLength( )

Return the length of array of bytes that are returned from the `digest( )` method. This value is usually constant (i.e., it does not depend on the amount of data that has been sent through the `update( )` method).

Let's see an example of how all of this works. As a simple case, let's say that we want to save a simple string to a file, but we're worried that the file might be corrupted when we read the string back in. Hence, in addition to saving the string, we must save a message digest. We do this by saving the serialized string object followed by the serialized array of bytes that constitute the message digest.

In order to save the pieces of data, we use this code:

```
package javasec.samples.ch11;

import java.io.*;
import java.security.*;

public class Send {
    public static void main(String args[]) {
        try {
            FileOutputStream fos = new FileOutputStream("test");
            MessageDigest md = MessageDigest.getInstance("SHA");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            String data = "This have I thought good to deliver thee, "+
               "that thou mightst not lose the dues of rejoicing " +
               "by being ignorant of what greatness is promised thee.";
            byte buf[] = data.getBytes(  );
            md.update(buf);
            oos.writeObject(data);
            oos.writeObject(md.digest(  ));
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

That's all there is to creating a digest of some data. The call to the `getInstance( )` method finds a message digest object that implements the SHA message digest algorithm. After creating our data −− which in this case is a simple string −− we pass that data to the `update( )` method of the message digest. In practice, this code could be slightly more complicated since all the data might not be available at once. As far as the message digest object is concerned, though, that situation would just require multiple calls to the `update( )` method instead of a single call (it can also be handled with digest streams, which we'll examine next). Once we've loaded all the data into the object, it is a simple matter to create the digest itself (with the `digest( )` method) and then save our data objects to the file.

Similarly, to retrieve this data we need only read the object back in and verify the message digest. In order to verify the message digest, we must recompute the digest over the data we received and test to make sure the digest is equivalent to the original digest:

```
package javasec.samples.ch11;

import java.io.*;
import java.security.*;

public class Receive {
    public static void main(String args[]) {
        try {
            FileInputStream fis = new FileInputStream("test");
            ObjectInputStream ois = new ObjectInputStream(fis);
            Object o = ois.readObject(  );
            if (!(o instanceof String)) {
                System.out.println("Unexpected data in file");
                System.exit(−1);
            }
            String data = (String) o;
            System.out.println("Got message " + data);
            o = ois.readObject(  );
            if (!(o instanceof byte[])) {
                System.out.println("Unexpected data in file");
                System.exit(−1);
            }
            byte origDigest[] = (byte []) o;
            MessageDigest md = MessageDigest.getInstance("SHA");
            md.update(data.getBytes(  ));
            if (MessageDigest.isEqual(md.digest(  ), origDigest))
                System.out.println("Message is valid");
            else System.out.println("Message was corrupted");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Once again, if the data was not available all at once, we would need to make multiple calls to the `update( )` method as the data arrived. We do not, however, need to make sure that calls to the `update( )` methods between the `Send` and `Receive` classes match in any sense; that is, if we called the `update( )` method four times in the `Send` class, we do not need to call the `update( )` method four times (with the same data) in the `Receive` class −− we can call it once, five times, or whatever. The calculation of the digest is unaffected by how the data was placed into the message digest object −− as long as the order of the bytes presented to the various calls to the `update( )` methods is the same.

11.2 Secure Message Digests

As we stated in Chapter 7, the message digest by itself does not give us a very high level of security. We can tell whether somehow the output file in this example has been corrupted because the text that we read in won't produce the same message digest that was saved with the file. But there's nothing to prevent someone from changing both the text and the digest stored in the file in such a way that the new digest reflects the altered text.

A secure message digest is called a Message Authentication Code (MAC). A MAC has the property that it cannot be created solely from the input data; it requires a secret key that is shared by the sender and receiver. Hence, an intermediate party cannot change both the data and the MAC without the receiver detecting that the data has been corrupted.

There are various ways in which a message digest can be made into a MAC, but the core Java security API does not provide any standard techniques for doing so. However, JCE does provide a class to produce a MAC, and there are simple ways to calculate a MAC on your own.

11.2.1 The Mac Class

The `Mac` class (`javax.crypto.Mac`) is part of the Java Cryptography Extension because it involves a cryptographic operation: a secret key is used to calculate the message digest. This mean that in order to use the `Mac` class, both sender and receiver must agree upon which secret key to use. As we discussed in Chapter 10, that means the sender and receiver must have previously exchanged and stored secret keys, either using a Diffie−Hellman key exchange or another means.

Sun's implementation of JCE provides two algorithms for calculating a MAC: `HmacSHA1` and `HmacMD5`, each of which is based on its respective message digest.

The interface to the `Mac` class is only slightly different than that of the `MessageDigest` class. There are still various update methods that are used to send data through the algorithm, but the `digest( )` method is replaced with the `doFinal( )` method:

public byte[] doFinal( )
public byte[] doFinal(byte[] input)
public void doFinal(byte[] output, int offset)

Calculate and return the MAC. The MAC is either returned directly or (in the last case) stored in the given array. The last two methods allow you to specify the last data to include in the MAC.

In order to calculate the size of the output array for the third method, use the `getMacLength( )` method.

In addition, before calling the `update( )` or `doFinal( )` methods, a MAC must be initialized by calling this method:

public void init(SecretKey sk)
public void init(SecretKey sk, AlgorithmParameterSpec aps)

Initialize the MAC for use. Failure to call this method will cause the `update( )` or `doFinal( )` method to throw an `IllegalStateException`.

The remainder of the methods of the `Mac` class mimic those in the `MessageDigest` class.

`Mac` objects can be reused any number of times; they are reset after each call to the `doFinal( )` method. They may be reused with a different key by calling the `init( )` method with the new key.

This modification of our last example saves a MAC instead of a simple digest:

```
package javasec.samples.ch11;

import java.io.*;
import java.security.*;
import javax.crypto.*;
import javasec.samples.ch10.KeyStoreHandler;

public class SendMac {
    public static void main(String args[]) {
        try {
            FileOutputStream fos = new FileOutputStream("test");
            Mac mac = Mac.getInstance("HmacSHA1");

            KeyStoreHandler ksh = new KeyStoreHandler(null);
            KeyStore ks = ksh.getKeyStore(  );
            mac.init((SecretKey) ks.getKey(args[0],
                                   args[1].toCharArray(  )));
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            String data = "This have I thought good to deliver thee, "+
               "that thou mightst not lose the dues of rejoicing " +
               "by being ignorant of what greatness is promised thee.";
            byte buf[] = data.getBytes(  );
            mac.update(buf);
            oos.writeObject(data);
            oos.writeObject(mac.doFinal(  ));
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Using standard key management techniques, we look up a secret key in the default keystore and use that key to initialize the `Mac` object. Otherwise, the code is very similar to what we've seen before. You can make similar changes to the `Receive` class so that it can read the MAC; we won't bother to show the code here (though it is included with the online examples). Note that you must have a secret key in your keystore to run this example; use the `StoreKey` example from Chapter 10 to create such a key.

11.2.2 Calculating Your Own MAC

A second way to create a MAC is to calculate one directly. This requires that both the sender and receiver of the data have a shared passphrase that they have kept secret, though that's often easier than sharing a secret key.

Using this passphrase, calculating a MAC requires that we:

1. Calculate the message digest of the secret passphrase concatenated with the data:

```
MessageDigest md = MessageDigest.getInstance("SHA");
String data = "This have I thought good to deliver thee, " +
                                "that thou mightst not lose the dues of rejoicing " +
                                "by being ignorant of what greatness is promised thee.";
String passphrase = "Sleep no more";
byte dataBytes[] = data.getBytes(  );
byte passBytes[] = passphrase.getBytes(  );
md.update(passBytes);
md.update(dataBytes);
byte digest1[] = md.digest(  );
```
2. Calculate the message digest of the secret passphrase concatenated with the just−calculated digest:

```
md.update(passBytes);
md.update(digest1);
byte mac[] = md.digest(  );
```

We can substitute this code in our original `Send` example, writing out the data string and the MAC to the file. Note that we can use the same message digest object to calculate both digests since the object is reset after a call to the `digest( )` method. Also note that the first digest we calculate is not saved to the file: we save only the data and the MAC. Of course, we must make similar changes to the `Receive` example; if the MACs are equal, the data was not modified in transit.

As long as we use exactly the same data for the passphrase in both the transmitting and receiving class, the message digests (that is, the MACs) still compare as equal. That gives a certain level of security to the message digest, but it requires that the sender and the receiver agree on what data to use for the passphrase; the passphrase cannot be transmitted along with the text. In this case, the security of the message digest depends upon the security of the passphrase. Normally, of course, you would prompt for that passphrase rather than hardcoding into the source as we've done above. In addition, a good passphrase would not be a well−known string such as we've selected; it would be random bytes (and hence indistinguishable from a secret key).

11.3 Message Digest Streams

The interface to the message digest class requires that you supply the data for the digest as a series of single bytes or byte arrays. As we mentioned earlier, this is not always the most convenient way to process data, which may be coming from a file or other input stream. This brings us to the message digest stream classes. These classes implement the standard input and output filter stream semantics of Java streams so that data can be written to a digest stream that will calculate the digest as the data itself is written (or the reverse operation for reading data).

Unfortunately, because the `Mac` class does not extend the `MessageDigest` class, these streams work only with standard message digests.

11.3.1 The DigestOutputStream Class

The first of these classes we'll examine is the `DigestOutputStream` class (`java.security.DigestOutputStream`). This class allows us to write data to a particular output stream and calculate the message digest of that data transparently as the data passes through the stream:

public class DigestOutputStream extends FilterOutputStream

Provide a stream that can calculate the message digest of data that is passed through the stream. A digest output stream holds two components internally: the output stream that is the ultimate destination of the data and a message digest object that computes the data of the stream written to the destination.

The digest output stream is constructed as follows:

public DigestOutputStream(OutputStream os, MessageDigest md)

Construct a digest output stream that associates the given output stream with the given message digest. Data that is written to the stream is automatically passed to the `update( )` method of the message digest.

In addition to the standard methods available to all output streams, a message digest output stream provides the following interface:

public MessageDigest getMessageDigest( )

Return the message digest associated with this output stream.

public void setMessageDigest(MessageDigest md)

Associate the given message digest with this output stream. The internal reference to the original message digest is lost, but the original message digest is otherwise unaffected (i.e., if you still hold a reference to the original message digest object, you can still calculate the digest of the data that was written to the stream while that digest was in place).

public void write(int b)
public void write(byte b[], int off, int len)

Write the given byte or array of bytes to the underlying output stream, and also update the internal message digest with the given data (if the digest stream is marked as on). These methods may throw an `IOException` from the underlying stream.

public void on(boolean on)

Turn the message digest stream on or off. When data is written to a stream that is off, the data will be passed to the underlying output stream, but the message digest will not be updated.

Note that this last method does not affect the underlying output stream at all; data is still sent to the underlying stream even if the digest output stream is marked as off. The on/off state only affects whether the `update( )` method of the message digest will be called as the data is written.

We can use this class to simplify the example we used earlier:

```
package javasec.samples.ch11;

import java.io.*;
import java.security.*;

public class SendStream {
    public static void main(String args[]) {
        try {
            FileOutputStream fos = new FileOutputStream("test");
            MessageDigest md = MessageDigest.getInstance("SHA");
            DigestOutputStream dos = new DigestOutputStream(fos, md);
            ObjectOutputStream oos = new ObjectOutputStream(dos);
            String data = "This have I thought good to deliver thee, "+
               "that thou mightst not lose the dues of rejoicing " +
               "by being ignorant of what greatness is promised thee.";
            oos.writeObject(data);
            dos.on(false);
            oos.writeObject(md.digest(  ));
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

The big change is in constructing the object output stream −− we now want to wrap it around the digest output stream so that as each object is written to the file, the message digest will include those bytes. We also want to make sure that we turn off the message digest calculation before we send the digest itself to the file. Turning off the digest isn't strictly necessary in this case since we don't use the digest object once we've calculated a single digest in this example, but it's good practice to keep the digest on only when strictly required.

Note that there is a subtle difference between the digest produced in this example and our first example. In the first example, the digest was calculated over just the bytes of the string that we saved to the file. In the second example, the digest was calculated over the serialized string object itself −− which includes some information regarding the class definition in addition to the bytes of the string.

11.3.2 The DigestInputStream Class

The symmetric operation to the digest output stream is the `DigestInputStream` class (`java.security.DigestInputStream`):

public class DigestInputStream extends FilterInputStream

Create an input stream that is associated with a message digest. When data is read from the input stream, it is also sent to the `update( )` method of the stream's associated message digest.

The digest input stream has essentially the same interface as the digest output stream (with writing replaced by reading). There is a single constructor for the class:

public DigestInputStream(InputStream is, MessageDigest md)

Construct a digest input stream that associates the given input stream with the given message digest. Data that is read from the stream will also automatically be passed to the `update( )` method of the message digest.

The interface provided by the digest input stream is symmetric to the digest output stream:

*public MessageDigest getMessageDigest( )**

Return the message digest that is associated with this output stream.

public void setMessageDigest(MessageDigest md)

Associate the given message digest with this output stream. The internal reference to the original message digest is lost, but the original message digest is otherwise unaffected (e.g., you can still calculate the digest of the data that had been written to the stream while that digest was in place).

public void read(int b)
public void read(byte b[], int off, int len)

Read one or more bytes from the underlying output stream, and also update the internal message digest with the given data (if the digest stream is marked as on). These methods may throw an `IOException` from the underlying stream.

public void on(boolean on)

Turn the message digest stream on or off. When data is read from a stream that is off, the message digest will not be updated.

Here's how we can use this class to read the file we created with the digest output stream:

```
package javasec.samples.ch11;

import java.io.*;
import java.security.*;

public class ReceiveStream {
    public static void main(String args[]) {
        try {
            FileInputStream fis = new FileInputStream("test");
            MessageDigest md = MessageDigest.getInstance("SHA");
            DigestInputStream dis = new DigestInputStream(fis, md);
            ObjectInputStream ois = new ObjectInputStream(dis);
            Object o = ois.readObject(  );
            if (!(o instanceof String)) {
                System.out.println("Unexpected data in file");
                System.exit(-1);
            }
            String data = (String) o;
            System.out.println("Got message " + data);
            dis.on(false);
            o = ois.readObject(  );
            if (!(o instanceof byte[])) {
                System.out.println("Unexpected data in file");
                System.exit(-1);
            }
            byte origDigest[] = (byte []) o;
            if (MessageDigest.isEqual(md.digest(  ), origDigest))
                System.out.println("Message is valid");
            else System.out.println("Message was corrupted");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Once again, constructing the input stream is a matter of providing a message digest. In this example, we've again turned off the digest input stream after reading the string object in the file. Turning off the stream is strictly required in this case. We want to make sure that the digest we calculate is computed only over the string object and not the stored byte array (that is, the stored message digest).

11.4 Implementing a MessageDigest Class

If you want to write your own security provider, you have the option of creating your own message digest engine. Typically, you'd do this because you want to ensure that a particular algorithm like SHA is available regardless of who the default security provider is; if you have a mathematics background, it's conceivable that you might want to implement your own algorithm.

In order to implement a message digest algorithm, you must provide a concrete subclass of the `MessageDigestSpi` class. That means providing a body for each of the following methods:

protected abstract void engineUpdate(byte input)
protected abstract void engineUpdate(byte[] input, int offset, int len)

Add the given bytes to the data over which the digest will be calculated. Note that there is no method in this list that accepts simply an array of bytes; the `update(byte[] b)` method in the base class simply uses an offset of and a length equal to the entire array.

protected abstract byte[] engineDigest( )

Calculate the digest over the accumulated data, resetting the internal state of the object afterwards. Note that there is no corresponding method that accepts an array of bytes as an argument; the `digest( )` method in the base class simply calls the `engineUpdate( )` method if needed before calling the `engineDigest( )` method.

protected int engineDigest(byte buf[], int offset, int len)

Calculate the digest, placing the output into the `buf` array (starting at the given `offset` and proceeding for `len` bytes) and returning the length of the calculated digest. The default implementation of this method simply calls the `engineDigest( )` method and then copies the result into `buf`. The buffer passed to this method always has sufficient length to hold the digest since if the buffer had been too short the `digest( )` method itself would have thrown an exception.

protected abstract void engineReset( )

Reset the internal state of the engine, discarding all accumulated data and resetting the algorithm to an initial condition.

protected int engineGetDigestLength( )

Return the digest length that is supported by this implementation. Unlike most of the protected methods in this class, this method is not abstract; it does not need to be overridden. However, the default implementation simply returns 0. If is returned by this method, the `getDigestLength( )` method attempts to create a clone of the digest object, calculate its digest, and return the length of the calculated digest. If a digest implementation does not override this method and does not implement the `Cloneable` interface, the `getDigestLength( )` method will not operate correctly.

Each of these methods corresponds to a public method of the `MessageDigest` class, with the name of the public method preceded by the word "engine". The public methods that do not have a corresponding method in this list are fully implemented in the base class and do not need to be implemented in the message digest subclass.

We'll show a simple implementation of a message digest class here. This implementation is based on a hash algorithm that produces a 4−byte output. As bytes are accumulated by this algorithm, they are stored into a 4−byte value (that is, an int); when this value has all four bytes filled, it is XOR−ed to another integer that accumulates the hash:

```
package javasec.samples.ch11;

import java.security.*;

public class XYZMessageDigest extends MessageDigest
                              implements Cloneable {
    private int hash;
    private int store;
    private int nBytes;

    public XYZMessageDigest(  ) {
        super("XYZ");
        engineReset(  );
    }

    public void engineUpdate(byte b) {
        switch(nBytes) {
            case 0:
                store =  (b << 24) & 0xff000000;
                break;
            case 1:
                store |= (b << 16) & 0x00ff0000;
                break;
            case 2:
                store |= (b <<  8) & 0x0000ff00;
                break;
            case 3:
                store |= (b <<  0) & 0x000000ff;
                break;
        }
        nBytes++;
        if (nBytes == 4) {
            hash = hash ^ store;
            nBytes = 0;
            store = 0;
        }
    }

    public void engineUpdate(byte b[], int offset, int length) {
        for (int i = 0; i < length; i++)
            engineUpdate(b[i + offset]);
    }

    public void engineReset(  ) {
        hash = 0;
        store = 0;
        nBytes = 0;
    }

    public byte[] engineDigest(  ) {
        while (nBytes != 0)
            engineUpdate((byte) 0);
        byte b[] = new byte[4];
        b[0] = (byte) (hash >>> 24);
        b[1] = (byte) (hash >>> 16);
        b[2] = (byte) (hash >>>  8);
        b[3] = (byte) (hash >>>  0);
        engineReset(  );
        return b;
    }
}
```

The implementation of this class is simple, which isn't surprising given the fact that the algorithm itself is too simple to be considered an effective digest algorithm. The major points to observe are:

- The name of the class (`XYZMessageDigest`) and the name of the algorithm that it implements (XYZ) must match one of the strings in the provider package for this class to be found. Hence, in our provider class in Chapter 8, we included this property:

  ```
  put("MessageDigest.XYZ", "XYZMessageDigest");
  ```

- Our constructor must be public and take no arguments. It must call the superclass constructor passing the name of the message digest algorithm it implements.
- In order for the `getDigestLength( )` method to function, we chose to implement the `Cloneable` interface instead of overriding the `engine−GetDigestLength( )` method. Since there are no embedded objects in this class, we do not need to override the `clone( )` method. The default implementation of that method (a shallow copy) is sufficient for this class.
- The `engineUpdate( )` methods accumulate bytes of data until an integer has been accumulated, at which point that integer can be XOR−ed into the saved state held in the `hash` instance variable.
- The `engineDigest( )` method converts the `hash` instance variable into a byte array and returns that to the programmer. Note that the `engineDigest( )` method is responsible for resetting the internal state of the algorithm. In addition, the `engineDigest( )` method is responsible for padding the data so that it is a multiple of four bytes (the size of a Java integer). This type of data padding is a common feature of message digest calculation.
- The `engineReset( )` method initializes the algorithm to its initial state.

Once we have an implementation of a message digest, we must install it into the security provider architecture. If we use the `XYZProvider` class from Chapter 8, we can change our `Send` class above to use our new digest algorithm:

```
package javasec.samples.ch11;

import java.io.*;
import java.security.*;
import javasec.samples.ch08.XYZProvider;

public class SendXYZ {
    public static void main(String args[]) {
        try {
            Security.addProvider(new XYZProvider(  ));
            FileOutputStream fos = new FileOutputStream("test.xyz");
            MessageDigest md = MessageDigest.getInstance("XYZ");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            String data = "This have I thought good to deliver thee, "+
               "that thou mightst not lose the dues of rejoicing " +
               "by being ignorant of what greatness is promised thee.";
            byte buf[] = data.getBytes(  );
            md.update(buf);
            oos.writeObject(data);
            oos.writeObject(md.digest(  ));
        } catch (Exception e) {
            System.out.println(e);
```

```
        }
    }
}
```

Similar changes to the `Receive` class will allow us to accept the message that we've saved to the file *test.xyz*.

11.4.1 The MacSpi Class

Implementing a MAC is similar, except that you extend the `MacSpi` class (`javax.crypto.MacSpi`). The method names within that class follow the method names of the `Mac` class: there is an `engineDoFinal( )` method instead of an `engineDigest( )` method, and so on. The key to the implementation of the `engineDoFinal( )` method is that it should use the key it received in the `engineInit( )` method to calculate a secure message digest (perhaps using the technique we outlined above when we used a simple passphrase).

Remember that the `Mac` class is a JCE engine, so the constructor of your MAC implementation must verify the JCE installation.

11.5 Comparison with Previous Releases

In Java 1.1, there is no `MessageDigestSpi` class, and the `MessageDigest` class simply extends `Object`. If you want to implement your own message digest in 1.1, then you extend the `MessageDigest` class directly. This does not cause any implementation differences.

The `getDigestLength( )` method does not exist in 1.1 nor does the `digest( )` method with a signature that accepts an array, offset, and length.

11.6 Summary

In this chapter, we've explored the message digest. The facility to calculate a message digest is straightforward and easy to use; the facility to write our own message digest class is equally straightforward.

The message digest by itself gives us some comfort about the state of the data it represents, but it does not give us a completely secure system. If we have a shared passphrase or secret key, we can construct a secure message digest (that is, a Message Authentication Code). A secure message digest is very similar to a digital signature, which we'll explore in the next chapter.

Chapter 12. Digital Signatures

In this chapter, we explore the mechanisms of the digital signature. The use and verification of digital signatures is another standard engine that is included in the security provider architecture. Like the other engines we've examined, the classes that implement this engine have both a public interface and an SPI for implementors of the engine.

We'll start by looking at the interface of the digital signature engine and see how you can create digitally signed objects that you can send between programs. We'll continue by looking into the details of digitally signed classes, including the `jarsigner` tool that creates those classes and how you can deal with those classes programatically. We'll conclude by looking at the details of the engine algorithm and how you can implement your own digital signature algorithms.

12.1 The Signature Class

When you handle digital signatures programatically, you perform two operations on them. You create them by taking a piece of data, creating a message digest of the data, and signing the message digest with a private key. The digitally signed data is then transmitted to someone else, who must verify the digital signature by creating a message digest of the data and verifying the signed digest using a public key. All of these operations are embodied within the `Signature` class (`java.security.Signature`):

public abstract class Signature extends SignatureSpi
Provide an engine to create and verify digital signatures.

The Sun security providers include implementations of this class that generate signatures based on the DSA and RSA algorithms.

12.1.1 Using the Signature Class

As with all engine classes, instances of the `Signature` class are obtained by calling one of these methods:

public static Signature getInstance(String algorithm)
public static Signature getInstance(String algorithm, String provider)
Generate a signature object that implements the given algorithm, optionally using the named provider. If an implementation of the given algorithm is not found, a `NoSuchAlgorithmException` is thrown. If the named security provider cannot be found, a `NoSuchProviderException` is thrown.

If the algorithm string is "DSA", the string "SHA/DSA" is substituted for it. Hence, implementors of this class that provide support for DSA signing must register themselves appropriately (that is, with the message digest algorithm name) in the security provider.

Once a signature object is obtained, the following methods can be invoked on it:

public void final initVerify(PublicKey publicKey)
Initialize the signature object, preparing it to verify a signature. A signature object must be initialized before it can be used. If the key is not of the correct type for the algorithm or is otherwise invalid, an `InvalidKeyException` is thrown.

public final void initSign(PrivateKey privateKey)

Initialize the signature object, preparing it to create a signature. A signature object must be initialized before it can be used. If the key is not of the correct type for the algorithm or is otherwise invalid, an `InvalidKeyException` is thrown.

public final void update(byte b)
public final void update(byte[] b)
public final void update(byte b[], int offset, int length)

Add the given data to the accumulated data the object will eventually sign or verify. If the object has not been initialized, a `SignatureException` is thrown.

public final byte[] sign( )
public final int sign(byte[] outbuf, int offset, int len)

Create the digital signature, assuming that the object has been initialized for signing. If the object has not been properly initialized, a `SignatureException` is thrown. Once the signature has been generated, the object is reset so that it may generate another signature based on some new data (however, it is still initialized for signing; a new call to the `initSign( )` method is not required).

In the first of these methods, the signature is returned from the method. Otherwise, the signature is stored into the `outbuf` array at the given offset, and the length of the signature is returned. If the output buffer is too small to hold the data, an `IllegalArgumentException` is thrown.

public final boolean verify(byte[] signature)

Test the validity of the given signature, assuming that the object has been initialized for verification. If the object has not been properly initialized, then a `SignatureException` is thrown. Once the signature has been verified (whether or not the verification succeeds), the object is reset so that it may verify another signature based on some new data (no new call to the `initVerify( )` method is required).

public final String getAlgorithm( )

Get the name of the algorithm this object implements.

public String toString( )

A printable version of a signature object is composed of the string `Signature object:`, followed by the name of the algorithm implemented by the object, followed by the initialized state of the object. The state is either `<not initialized>`, `<initialized for verifying>`, or `<initialized for signing>`. However, the Sun implementations of this class override this method to show the parameters of the algorithm instead.

public final void setParameter(String param, Object value) [deprecated]
public final void setParameter(AlgorithmParameterSpec param)

Set the parameter of the signature engine. In the first format, the named parameter is set to the given value; in the second format, parameters are set based on the information in the `param` specification.

In the Sun implementation of the DSA signing algorithm, the only valid `param` string is `KSEED`, which requires an array of bytes that will be used to seed the random number generator used to generate the `k` value. There is no way to set this value through the parameter specification, which in the Sun implementation always returns an `UnsupportedOperationException`.

public final Object getParameter(String param) [deprecated]

Return the named parameter from the object. The only valid string for the Sun implementation is `KSEED`.

public final Provider getProvider( )

Return the provider that supplied the implementation of this signature object.

It is no accident that this class has many similarities to the `MessageDigest` class; a digital signature algorithm is typically implemented by performing a cryptographic operation combining a private key and the message digest that represents the data to be signed. For the developer, this means generating a digital signature is virtually the same as generating a message digest; the only difference is that a key must be presented in order to operate on a signature object. This difference is important, however, since it fills in the hole we noticed previously: a message digest can be altered along with the data it represents so that the tampering is unnoticeable. A signed message digest, on the other hand, can't be altered without knowledge of the key that was used to create it. The use of a public key in the digital signature algorithm makes the digital signature more attractive than a message authentication code in which there must be a shared key between the parties involved in the message exchange.

Let's take our example from Chapter 11, where we saved a message and its digest to a file; we'll modify it now to save the message and the digital signature. We can create the digital signature like this:

```
package javasec.samples.ch12;

import java.io.*;
import java.security.*;
import javasec.samples.ch10.KeyStoreHandler;

public class Send {
    public static void main(String args[]) {
        String data;
        data = "This have I thought good to deliver thee, " +
               "that thou mightst not lose the dues of rejoicing " +
               "by being ignorant of what greatness is promised thee.";

        try {
            FileOutputStream fos = new FileOutputStream("test");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            KeyStoreHandler ksh = new KeyStoreHandler(null);
            KeyStore ks = ksh.getKeyStore(  );
            PrivateKey pk = (PrivateKey) ks.getKey(args[0],
                                   args[1].toCharArray(  ));

            Signature s = Signature.getInstance("MD5withRSA");
            s.initSign(pk);

            byte buf[] = data.getBytes(  );
            s.update(buf);
            oos.writeObject(data);
            oos.writeObject(s.sign(  ));
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

This example puts together many of the examples from the past few chapters. In order to create the digital signature we must accomplish the following:

1. Obtain the private key that is used to sign the data. Here we're using the keystore handler we wrote previously and the command−line arguments to obtain the alias and password of the private key we want to use.
2. Obtain a signing object via the `getInstance( )` method and initialize it. Since we're creating a signature in this example, we use the `initSign( )` method for initialization.
3. Pass the data to be signed as a series of bytes to the `update( )` method of the signing object. Multiple calls could be made to the `update( )` method even though in this example we only need one.
4. Obtain the signature by calling the `sign( )` method. We save the signature bytes and write them to a file with the data so that the data and the signature can be retrieved at a later date.

Reading the data and verifying the signature are similar:

```
package javasec.samples.ch12;

import java.io.*;
import java.security.*;
import javasec.samples.ch10.KeyStoreHandler;

public class Receive {
    public static void main(String args[]) {
        try {
            String data = null;
            byte signature[] = null;
            FileInputStream fis = new FileInputStream("test");
            ObjectInputStream ois = new ObjectInputStream(fis);
            Object o = ois.readObject(  );
            try {
                data = (String) o;
            } catch (ClassCastException cce) {
                System.out.println("Unexpected data in file");
                System.exit(−1);
            }
            o = ois.readObject(  );
            try {
                signature = (byte []) o;
            } catch (ClassCastException cce) {
                System.out.println("Unexpected data in file");
                System.exit(−1);
            }
            System.out.println("Received message");
            System.out.println(data);

            KeyStoreHandler ksh = new KeyStoreHandler(null);
            KeyStore ks = ksh.getKeyStore(  );

            java.security.cert.Certificate c =
                                    ks.getCertificate(args[0]);
            PublicKey pk = c.getPublicKey(  );
            Signature s = Signature.getInstance("MD5withRSA");
            s.initVerify(pk);
            s.update(data.getBytes(  ));
            if (s.verify(signature)) {
                System.out.println("Message is valid");
            }
            else System.out.println("Message was corrupted");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

The process of verifying the signature still requires four steps. The major differences are that in step two, we initialize the signing object for verification by using the `initVerify( )` method, and in step four, we verify (rather than create) the existing signature by using the `verify( )` method. Note that we still have to know who signed the message in order to look up the correct key -- but more about that a little later.

12.1.2 The SignedObject Class

In our last example, we had to create an object that held both the data in which we are interested and the signature for that data. This is a common enough requirement that Java provides the `SignedObject` class (`java.security.SignedObject`) to encapsulate an object and its signature:

public final class SignedObject implements Serializable

Encapsulate an object and its digital signature. The encapsulated object must be serializable so that a serialization of a signed object can do a deep copy of the embedded object.

Signed objects are created with this constructor:

public SignedObject(Serializable o, PrivateKey pk, Signature engine)

Create a signed object based on the given object, signing the serialized data in that object with the given private key and signature object. The signed object contains a copy of the given object; this copy is obtained by serializing the object parameter. If this serialization fails, an `IOException` is thrown.

It's very important to realize that this constructor makes, in effect, a copy of its parameter; if you create a signed object based on a string buffer and later change the contents of the string buffer, the data in the signed object remains unchanged. This preserves the integrity of the object encapsulated with its signature.

Here are the methods we can use to operate on a signed object:

public Object getContent( )

Return the object embedded in the signed object. The object is reconstituted using object serialization; an error in serialization may cause either an `IOException` or a `ClassNotFoundException` to be thrown.

public byte[] getSignature( )

Return the signature embedded in the signed object.

public String getAlgorithm( )

Return the name of the algorithm that was used to sign the object.

public boolean verify(PublicKey pk, Signature s)

Verify the signature within the embedded object with the given key and signature engine. The signature engine parameter may be obtained by calling the `getInstance( )` method of the `Signature` class. The underlying signature engine may throw an `InvalidKeyException` or `SignatureException`.

We'll use this class in examples later in this chapter.

12.1.3 Signing and Certificates

In the previous examples, we specified on the command line the name of the entity that we assumed generated the signature in the file. This was necessary because the file contained only the actual signature of the entity and the data that was signed; it did not contain any information about who the signer actually is. That's fine for an example, but it is not always appropriate in a real application. We could have asked the user for the name of the entity that was supposed to have signed the data, but that course is fraught with potential errors:

- The user could have no idea what names are in the keystore of the application. Especially in a corporate environment, users may not know what data the keystore database might contain.
- The user could get the name of the keystore alias wrong. Say that the application asks the user to enter the name of the signer; the user, knowing that the data came from me, may enter "sdo" as the alias of the identity.

  What the user may not remember is that when the keystore was first created, she received a public key from the San Diego Oil company; that public key was entered into the keystore with the alias "sdo." When my identity was added to the keystore, a different alias had to be chosen, so my public key was added with the alias "ScottOaks." But that was a long time ago, now forgotten, and because I use the `sdo` moniker all over my writings, the user assumes that I am the `sdo` in the keystore. And so the wrong alias will be chosen, and the signature verification will fail when it should have succeeded.

For these reasons, it makes more sense to include the public key with the signature and the signed data. This allows the application to find the identity based on the unique public key in order to determine who the signer of the data is.

We could do that by simply sending the encoded public key with the signature and data. A better solution, however, would be to send the certificate that verifies the public key. That way, if the public key is not found in the database, the credentials of the certificate can be presented to the user, and the user can have the opportunity to decide on the fly if the particular entity should be trusted.

Although an embedding of signature, data, and certificate is very common, the `SignedObject` class does not include the capability to contain a certificate. So we'll use the `SignedObject` class in this example, but we'll still need an object that contains the signed object and the certificate. We'd like to do this by extending the `SignedObject` class, but since that class is `final` we're forced to adopt this approach:

```
package javasec.samples.ch12;

import java.io.*;
import java.security.*;
import java.security.cert.*;

public class Message implements Serializable {
    SignedObject object;
    transient java.security.cert.Certificate certificate;

    private void writeObject(ObjectOutputStream out)
                                    throws IOException {
```

```
        out.defaultWriteObject(  );
        try {
            out.writeObject(certificate.getEncoded(  ));
        } catch (CertificateEncodingException cee) {
            throw new IOException("Can't serialize object " + cee);
        }
    }

    private void readObject(ObjectInputStream in)
                           throws IOException, ClassNotFoundException {
        in.defaultReadObject(  );
        try {
            byte b[] = (byte []) in.readObject(  );
            CertificateFactory cf =
                        CertificateFactory.getInstance("X509");
            certificate = cf.generateCertificate(new
                             ByteArrayInputStream(b));
        } catch (CertificateException ce) {
            throw new IOException("Can't de−serialize object " + ce);
        }
    }
}
```

We've made the `certificate` variable in this class transient and have explicitly serialized and deserialized it using its external encoding. As we discussed in Chapter 9, whenever we have an embedded certificate or key, we should follow a procedure like this to ensure that the receiving party is able to deserialize the class.

As it turns out, the X509 certificate implementation that comes with the SDK (that is, the `sun.security.x509.X509CertImpl` class) also overrides the `writeObject( )` and `readObject( )` methods, so if we serialize a certificate explicitly, the encoded data is written to or read from the file. It is not sufficient to rely upon that, however −− if we use the default serialization methods for the `Message` class, a reference to the `sun.security.x509.X509CertImpl` class is embedded into the serialized stream. A user with another security provider (and hence a different implementation of the `X509Certificate` class) would not be able to deserialize the stream because there is no access to the Sun implementation of the `X509Certificate` class. Explicitly serializing and deserializing the certificate as we've done here avoids embedding any reference to the provider class and makes the data file more portable.

When we save the message to the file, we now have to make sure that we save a certificate with it. Other than that, changes to the class are minor:

```
package javasec.samples.ch12;

import java.io.*;
import java.security.*;
import javasec.samples.ch10.KeyStoreHandler;

public class SendObject {
    public static void main(String args[]) {
        try {
            FileOutputStream fos = new FileOutputStream("test.obj");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            KeyStoreHandler ksh = new KeyStoreHandler(null);
            KeyStore ks = ksh.getKeyStore(  );

            java.security.cert.Certificate certs[] =
                               ks.getCertificateChain(args[0]);
            PrivateKey pk = (PrivateKey) ks.getKey(args[0],
                                         args[1].toCharArray(  ));
            Message m = new Message(  );
            m.object = new SignedObject(
              "This have I thought good to deliver thee, " +
              "that thou mightst not lose the dues of rejoicing " +
              "by being ignorant of what greatness is promised thee.",
                            pk, Signature.getInstance("MD5withRSA"));
            m.certificate = certs[0];
            oos.writeObject(m);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Essentially, the only change we have made is to use the new `Message` class to store the data that we're sending.

Retrieving the data is now more complicated, since we must verify both the signature in the signed object and the identity of the authority that signed the embedded certificate:

```
package javasec.samples.ch12;

import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
import javasec.samples.ch10.KeyStoreHandler;

public class ReceiveObject {
    private static void verifySigner(java.security.cert.Certificate c,
                        String name) throws CertificateException {
        java.security.cert.Certificate issuerCert = null;
        X509Certificate sCert = null;
        KeyStore ks = null;

        try {
            KeyStoreHandler ksh = new KeyStoreHandler(null);
            ks = ksh.getKeyStore(  );
        } catch (Exception e) {
            throw new CertificateException("Invalid keystore");
        }

        try {
            String signer = ks.getCertificateAlias(c);
            if (signer !=null){
                System.out.println("We know the signer as " + signer);
                return;
            }
            for (Enumeration alias = ks.aliases(  );
                              alias.hasMoreElements(  );){
                String s = (String) alias.nextElement(  );
                try {
                    sCert = (X509Certificate) ks.getCertificate(s);
                } catch (Exception e) {
                    continue;
                }
                if (name.equals(sCert.getSubjectDN().getName(  ))){
                    issuerCert = sCert;
                    break;
                }
            }
        } catch(KeyStoreException kse) {
            throw new CertificateException("Invalid keystore");
        }
        if (issuerCert == null) {
            throw new CertificateException("No such certificate");
        }
        try {
            c.verify(issuerCert.getPublicKey(  ));
        } catch (Exception e) {
            throw new CertificateException(e.toString(  ));
        }
    }

    private static void processCertificate(X509Certificate x509)
                               throws CertificateParsingException {
        Principal p;
        p = x509.getSubjectDN(  );
        System.out.println("This message was signed by " +
                           p.getName(  ));
        p = x509.getIssuerDN(  );
        System.out.println("This certificate was provided by " +
                           p.getName(  ));
        try {
            verifySigner(x509, p.getName(  ));
        } catch (CertificateException ce) {
            System.out.println("We don't know the certificate signer");
        }
        try {
            x509.checkValidity(  );
        } catch (CertificateExpiredException cee) {
            System.out.println("That certificate is no longer valid");
        } catch (CertificateNotYetValidException cnyve) {
            System.out.println("That certificate is not yet valid");
        }
    }

    public static void main(String args[]) {
        try {
            FileInputStream fis = new FileInputStream("test.obj");
            ObjectInputStream ois = new ObjectInputStream(fis);
            Object o = ois.readObject(  );
            if (o instanceof Message) {
                Message m = (Message) o;
                System.out.println("Received message");
                processCertificate((X509Certificate) m.certificate);
                PublicKey pk = m.certificate.getPublicKey(  );
                if (m.object.verify(pk,
                              Signature.getInstance("MD5withRSA"))) {
                    System.out.println("Message is valid");
                    System.out.println(m.object.getObject(  ));
                }
                else System.out.println("Signature is invalid");
            }
            else System.out.println("Message is corrupted");
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

We've seen most of this code in previous chapters; in particular, the `processCertificate( )` method uses the standard certificate methods to extract and print information about the certificate. The new code for us is primarily in the `verifySigner( )` method, where we search the entire keystore for a name that matches the issuer of the certificate that was sent to us. If we find a match, we use the corresponding public key to verify the certificate we received.

This method shows yet another need for an alternate implementation of the `KeyStore` class -- if you have to search the entire list of keys for a matching certificate like this, you clearly don't want to perform a linear search each time. An alternate keystore could provide a more efficient means of searching for certificates.

12.2 Signed Classes

One of the primary applications of digital signatures in Java is to create and verify signed classes. Signed classes allow the expansion of Java's sandbox in two ways:

- The policy file can insist that classes coming from a particular site be signed by a particular entity before the access controller will grant that particular set of permissions. In the policy file, such an entry contains a `signedBy` directive:

  ```
  grant signedBy "sdo", codeBase "http://piccolo.East.Sun.COM/" {
                  java.io.FilePermission "-", "read,write";
  }
  ```

  This entry allows classes that are loaded from *piccolo.East.Sun.COM* to read and write any local files under the current directory only if the classes have been signed by `sdo`.
- The security manager can cooperate with the class loader in order to determine whether or not a particular class is signed; the security manager is then free to grant permissions to that class based on its own internal policy. However, while this was an important technique in Java 1.1, it is rarely used in Java 2.

We talked about these operations throughout this book; in this section, we'll fill in the last details about how the digital signatures are created and verified. There are three necessary ingredients to expand the Java sandbox with signed classes:

- A method to create the signed class. The `jarsigner` utility is used for this.
- A class loader that knows how to understand the digital signature associated with the class. The `URLClassLoader` class knows how to do this, but we'll show an example of how to do that for a custom class loader as well.
- A security manager or access controller that grants the desired permissions based on the digital signature. The access controller will do this for us; we'll show how the security manager might do this directly in Appendix D.

12.2.1 The jarsigner Tool

Signed jar files are managed with `jarsigner`. `jarsigner` uses the information in a keystore to look up information about a particular entity and uses that information either to sign or to verify a jar file. As we discussed in Chapter 10, the keystore that `jarsigner` uses is subject to the `KeyStore` class that has been installed into the virtual machine; if you have your own keystore implementation, `jarsigner` will be able to use it. Similarly, if you use the standard keystore implementation but hold the keys in a file other than the default *.keystore* file, `jarsigner`will allow you to use that other file as well.

A signed jar file is identical to a standard jar file except that a signed jar file contains two additional entries:

SIGNER.SF

A file containing an SHA message digest for each class file in the archive. The digest is calculated from the three lines in the manifest for the class file. The base of this name (`SIGNER`) varies; it is typically based upon the alias of the keystore entry used to sign the archive.

SIGNER.DSA

A file containing the digital signature of the `.SF` file. The base of this name matches the first part of the `.SF` file; the extension is the algorithm used to generate the signature. This file also contains the certificate of the entity that signed the archive.

The algorithm used to generate the signature depends upon the type of the key found in the keystore: if the key is an X509 (DSA) key, a DSA signature will be generated. If the key is an RSA key, an RSA signature will be generated.

These entries are held in the *META−INF* directory of the jar file.

12.2.1.1 Creating a signed jar file

The simplest command to sign a jar file is:

```
piccolo% jarsigner xyz.jar sdo
```

This command takes the existing jar file *xyz.jar* and signs it using the private key of the given alias (`sdo`). The private key is obtained by searching for the given alias from the default keystore (*$HOME/.keystore*). The signature files in this example will be named *SDO.SF* and *SDO.DSA* and will be added to the existing jar file.

A jar file can be signed by any number of entities simply by executing this command multiple times with different aliases. Each act of signing the jar file creates a new set of `.SF` and `.DSA` or `.RSA` files in the archive.

A number of options can be used in conjunction with this command:

−keystore keystore

Specify the file that should be used as the keystore.

−storepass storepass

Specify the global password that should be used to open the keystore. If this value is not provided, you will be prompted for it (which, as always, is the more secure way to enter a password).

−storetype storetype

The algorithm type of the keystore (e.g., JCEKS).

−keypass password

Specify the password for the key entry of the given alias. If this value is not provided, you will be prompted for it.

−sigfile file

Specify the base name to be used for the `.SF` and `.DSA/.RSA` files. The default for this value is the alias specified on the command line translated to all uppercase letters (e.g., *SDO* for the example above). If the alias name has more than eight letters, only the first eight letters are used. The file argument in this option can only contain uppercase letters, the digits 0−9, and an underscore; it must contain eight or fewer letters.

−signedjar file
Write the signed jar file to the named file instead of adding the signature entries to the existing jar file.

−internalsf
Include the `.SF` file in the `.DSA/.RSA` file. This is present for backward compatibility and should not be used since it increases the size of the jar file.

−sectionsonly
Don't include header information that can be used to speed up verification. This is present for backward compatibility and should not be used since it requires longer to verify the jar file.

−verbose
Print out information as `jarsigner` progresses.

12.2.1.2 Verifying a jar file

`Jarsigner` can also verify a jar file. In the process of verifying a jar file, `jarsigner` will use the public key of the certificate embedded in the jar file to verify that the signature is valid. The simplest command to verify a jar file is:

```
piccolo% jarsigner -verify xyz.jar
jar verified.
```

If the jar file can't be verified (because it has been corrupted), an error message with a Java exception will be printed instead.

Verification accepts the following options:

−sigfile file
Use the given base name to look up the `.SF` and `.DSA/.RSA` files. This option is useful when the jar file has been signed by multiple entities.

−verbose
Provide verbose output for the verification, indicating for each file if it was signed and whether or not the signer of the file has been found in the keystore. Sample output from this command might appear like this:

```
piccolo% jarsigner -verify -verbose xyz.jar

        402 Mon Jan 26 19:25:52 EST 1998 META-INF/SDO.SF
       1395 Mon Jan 26 19:25:52 EST 1998 META-INF/SDO.DSA
smk     596 Sat Jan 24 22:18:22 EST 1998 XYZKey.class
smk     814 Sat Jan 24 22:17:46 EST 1998 XYZKeyPairGenerator.class
smk    1155 Sat Jan 24 21:56:40 EST 1998 XYZProvider.class
smk     900 Sat Jan 24 22:11:22 EST 1998 XYZSignature.class

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.
```

Note the legend for each file that is printed by this command. We know if the file was signed, whether or not it was listed in the jar file's manifest, and whether or not the signer of the file was found in the keystore.

In the vast majority of cases, the information for each file will be the same: jar files are usually signed all at once by the same person. However, there's nothing to prevent someone from adding a new class to a signed jar file (in which case the class would appear as unsigned) or for a jar file to contain multiple signers (some of whom may have signed some of the classes, while others may have signed only a few of the classes).

In order to determine whether the certificate was found in the keystore, `jarsigner` opens the default instance of the `KeyStore` class and loads it. Note that no password is required for this operation; we're reading only public information from the keystore.

−certs

In conjunction with the `−verbose` option, print out the certificates (if any) that are found with each class. With this option, the output for a particular class looks like this:

```
smk     900 Sat Jan 21 22:11:22 EST 2001 XYZSignature.class
      X.509, EmailAddress=scott.oaks@sun.com, CN=Thawte Freemail Member
      X.509, CN=Personal Freemail RSA 2000.8.30, \
             OU=Certificate Services, O=Thawte, L=Cape Town, \
             ST=Western Cape, C=ZA
      X.509, EmailAddress=personal−freemail@thawte.com, \
             CN=Thawte Personal Freemail CA, \
             OU=Certification Services Division, O=Thawte Consulting,
             L=Cape Town, ST=Western Cape, C=ZA
```

In this case, the class was signed by the given distinguished name (Thawte Freemail Member with an EmailAddress of scott.oaks@sun.com); the certificate chain shows who issued each certificate. This information is repeated for each signed class.

This option has no effect unless the `−verbose` option is specified.

−keystore keystore

Use the given file as the name of the keystore to load. This name is only used for the `−verbose` option to look up the certificates of the signer.

12.2.2 Reading Signed Jar Files

As we've just seen, a signed jar file has three special elements:

- A manifest (*MANIFEST.MF*), containing a listing of the files in the archive that have been signed, along with a message digest for each signed file.
- A signature file (*XXX.SF*, where *XXX* is the name of the entity that signed the archive) that contains signature information. The data in this file is comprised of message digests of entries in the manifest file.
- A block file (*XXX.DSA*, where *XXX* is the name of the entity that signed the archive and DSA is the name of the signature algorithm used to create the signature). The block file contains the actual signature data in a format known as PKCS7.

There are many advantages to this format, not the least of which is that the PKCS7 block file (that is, the signature itself) is a standard format for external signatures. Unfortunately, the necessary classes to create PKCS7 blocks are not part of Java's public API; if you want to be able to write a signed jar file, you'll need to write the classes to create the signature block yourself.

However, we can read a signed jar file using the core API. In Chapter 6, when we created a code source we always passed `null` for the array of certificates that had signed the class. We'll extend that class loader now so that it handles reading the correct certificates from the signed jar file and uses those certificates to create an appropriate code source.

Note that the URL class loader already does this for you; you need this technique only when you extend the `SecureClassLoader` class directly. As it happens, the classes in the `java.util.jar` package can handle reading the signature file for us, so all we need to do is to write a class loader that uses the jar−handling classes in that package and that creates the correct code source using certificates retrieved by those classes:

```
package javasec.samples.ch12;

import java.io.*;
import java.net.*;
import java.security.*;
import java.util.*;
import java.util.jar.*;

public class JarLoader extends SecureClassLoader {
    private URL urlBase;
    public boolean printLoadMessages = true;

    // These hold the classes and certificates that we read from
    // the jar file
    Hashtable classArrays;
    Hashtable classIds;

    // Construct the JarLoader. The base is the URL from which we expect to
    // read classes (e.g., http://piccolo/Car.class is read from
    // http://piccolo/). In addition, we can specify jar files to be read
    // by calling the readJarFile(  ) method with the name of the jar file
    // relative to the base. So we can read both jar files and individual
    // class files from this class loader.
    public JarLoader(String base, ClassLoader parent) {
        super(parent);
        try {
            if (!(base.endsWith("/")))
                base = base + "/";
            urlBase = new URL(base);
            classArrays = new Hashtable(  );
            classIds = new Hashtable(  );
        } catch (Exception e) {
            throw new IllegalArgumentException(base);
        }
    }

    // Completely read the input stream and convert it into
    // a byte array.
    private byte[] getClassBytes(InputStream is) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream(  );
        BufferedInputStream bis = new BufferedInputStream(is);
        boolean eof = false;
        while (!eof) {
            try {
                int i = bis.read(  );
                if (i == −1)
                    eof = true;
                else baos.write(i);
            } catch (IOException e) {
                return null;
            }
        }
        return baos.toByteArray(  );
    }

    // Find the class. This is based on the steps of our
    // previous example in Chapter 6.
    protected Class findClass(final String name)
                              throws ClassNotFoundException {
        try {
            return (Class)
                AccessController.doPrivileged(
                    new PrivilegedExceptionAction(  ) {
                        public Object run(  )
                                      throws ClassNotFoundException {
                            Object o = doFindClass(name);
                            if (o == null)
                                throw new ClassNotFoundException(name);
                            return o;
                        }
                    }
                );
        } catch (PrivilegedActionException pae) {
            throw (ClassNotFoundException) pae.getException(  );
        }
    }

    // This does the work of loading the class. There are two places
    // from which we can load classes: we can check the classArray
    // hashtable for things we've read from a jar file; those classes
    // might have been signed in which case the signing certificates
    // will be in the classIds hashtable. The second place we can look
    // is for an individual class file from the url base (in which case
    // it will be unsigned).
    private Class doFindClass(String name) {
        String urlName = name.replace('.', '/');
        byte buf[];
        Class cl;

        SecurityManager sm = System.getSecurityManager(  );
        if (sm != null) {
            int i = name.lastIndexOf('.');
            if (i >= 0)
                sm.checkPackageDefinition(name.substring(0, i));
        }

        // See if the class is in a jar file we've read.
        buf = (byte[]) classArrays.get(urlName);
        if (buf != null) {
            // It was -- get the certificates (if any) and define the
            // codesource and class with them.
            java.security.cert.Certificate[] ids =
              (java.security.cert.Certificate[]) classIds.get(urlName);
            CodeSource cs = new CodeSource(urlBase, ids);
            cl = defineClass(name, buf, 0, buf.length, cs);
            return cl;
        }

        // The class wasn't in a jar file -- see if we can load it
        // directly. Since it's being loaded directly, it won't have
        // a signature.
        try {
            URL url = new URL(urlBase, urlName + ".class");
            if (printLoadMessages)
                System.out.println("Loading " + url);
            InputStream is = url.openConnection().getInputStream(  );
            buf = getClassBytes(is);
            CodeSource cs = new CodeSource(urlBase, null);
            cl = defineClass(name, buf, 0, buf.length, cs);
            return cl;
        } catch (Exception e) {
            System.out.println("Can't load " + name + ": " + e);
            return null;
        }
    }

    // Read a jar file, storing its classes into the classArray and
    // the certificates of its signatures into the classIds.
    public void readJarFile(String name) {
        URL jarUrl = null;
        JarInputStream jis;
        JarEntry je;

        try {
            jarUrl = new URL(urlBase, name);
        } catch (MalformedURLException mue) {
            System.out.println("Unknown jar file " + name);
            return;
        }
        if (printLoadMessages)
            System.out.println("Loading jar file " + jarUrl);

        try {
            jis = new JarInputStream(
                        jarUrl.openConnection().getInputStream(  ));
        } catch (IOException ioe) {
            System.out.println("Can't open jar file " + jarUrl);
            return;
        }

        try {
            while ((je = jis.getNextJarEntry(  )) != null) {
                String jarName = je.getName(  );
                if (jarName.endsWith(".class"))
                    loadClassBytes(jis, jarName, je);
                // else ignore it; it could be an image or audio file
                // Really, these type of entries need to be saved for
                // the resource methods; we leave that extension to
                // the reader.
                jis.closeEntry(  );
            }
        } catch (IOException ioe) {
            System.out.println("Badly formatted jar file");
        }
    }

    private void loadClassBytes(JarInputStream jis,
                                String jarName, JarEntry je) {
        if (printLoadMessages)
            System.out.println("\t" + jarName);
        BufferedInputStream jarBuf = new BufferedInputStream(jis);
        ByteArrayOutputStream jarOut = new ByteArrayOutputStream(  );
        int b;
        try {
            while ((b = jarBuf.read(  )) != -1)
                jarOut.write(b);
            String className = jarName.substring(0,
                                                 jarName.length(  ) - 6);
            classArrays.put(className, jarOut.toByteArray(  ));
            // This returns the certificates from the signature file
            // only if this class was signed.
            java.security.cert.Certificate c[] = je.getCertificates(  );
            if (c == null)
                c = new java.security.cert.Certificate[0];
            classIds.put(className, c);
        } catch (IOException ioe) {
            System.out.println("Error reading entry " + jarName);
        }
    }
}
```

Although it's a long example, most of it simply deals with reading entries from the jar file. All that we're left to do from a security perspective is obtain the array of signers when we read in each jar entry and then use that array of signers when we construct the code source we use to define the class. Remember that each file in a jar file may be signed by a different group of identities and that some may not be signed at all. This is why we must construct a new code source object for each signed class that was in the jar file.

12.3 Implementing a Signature Class

Now that we've seen how to use the `Signature` class, we'll look at how to implement our own class. The techniques we'll see here should be very familiar from our other examples of implementing an engine in the security provider architecture. In particular, since in Java 2 the `Signature` class extends its own SPI, we can implement a single class that extends the `Signature` class.

To construct our subclass, we must use the following constructor:

protected Signature(String algorithm)

This is the only constructor of the `Signature` class, so all subclasses of this class must use this constructor. The string passed to the constructor is the name that will be registered with the security provider.

Once we've constructed our engine object, we must implement the following methods in it:

protected abstract void engineInitVerify(PublicKey pk)

Initialize the object to prepare it to verify a digital signature. If the public key does not support the correct algorithm or is otherwise corrupted, an `InvalidKeyException` is thrown.

protected abstract void engineInitSign(PrivateKey pk)

Initialize the object to prepare it to create a digital signature. If the private key does not support the correct algorithm or is otherwise corrupted, an `InvalidKeyException` is thrown.

protected abstract void engineUpdate(byte b)
protected abstract void engineUpdate(byte b[], int off, int len)

Add the given bytes to the data that is being accumulated for the signature. These methods are called by the `update(  )` methods; they typically call the `update(  )` method of a message digest held in the engine. If the engine has not been correctly initialized, a `SignatureException` is thrown.

protected abstract byte[] engineSign( )
protected int engineSign(byte[] outbuf, int offset, int len)

Create the signature based on the accumulated data. If there is an error in generating the signature, a `SignatureException` is thrown.

protected abstract boolean engineVerify(byte b[])

Return an indication of whether or not the given signature matches the expected signature of the accumulated data. If there is an error in validating the signature, a `SignatureException` is thrown.

protected abstract void engineSetParameter(String p, Object o) [deprecated]
protected abstract void engineSetParameter(AlgorithmParameterSpec p)

Set the given parameters, which may be algorithm−specific. If this parameter does not apply to this algorithm, this method should throw an `InvalidParameterException`.

protected abstract Object engineGetParameter(String p) [deprecated]

Return the desired parameter, which is algorithm−specific. If the given parameter does not apply to this algorithm, this method should throw an `InvalidParameterException`.

In addition to those methods, there are a few protected instance variables that keep track of the state of the signature object −− whether it has been initialized, whether it can be used to sign or to verify, and so on:

protected final static int UNINITIALIZED
protected final static int SIGN
protected final static int VERIFY
protected int state

These variables control the internal state of signature object. The state is initially `UNINITIALIZED`; it is set to `SIGN` by the `initSign( )` method and to `VERIFY` by the `initVerify( )` method.

These variables are not normally used by the subclasses of `Signature` since the logic to maintain them is already implemented in the `Signature` class itself.

Here is an implementation of a signature class. Note that the `XYZSign` class depends on other aspects of the security architecture −− in this example, the message digest engine to create an SHA message digest and the DSA key interfaces to handle the public and private keys. This is very typical of signature algorithms −− even to the point where the default name of the algorithm reflects the underlying components. The actual encryption of the message digest will use a simple XOR−based algorithm (so that we can, as usual, avoid the mathematics involved with a secure example):

```
package javasec.samples.ch12;

import java.security.*;
import java.security.interfaces.*;
import java.security.spec.*;

public class XYZSignature extends Signature implements Cloneable {
    private DSAPublicKey pub;
    private DSAPrivateKey priv;
    private MessageDigest md;

    public XYZSignature( ) throws NoSuchAlgorithmException {
        super("XYZSignature");
        md = MessageDigest.getInstance("SHA");
    }

    public void engineInitVerify(PublicKey publicKey)
                                throws InvalidKeyException {
        try {
            pub = (DSAPublicKey) publicKey;
        } catch (ClassCastException cce) {
            throw new InvalidKeyException("Wrong public key type");
        }
    }

    public void engineInitSign(PrivateKey privateKey)
                                throws InvalidKeyException {
        try {
            priv = (DSAPrivateKey) privateKey;
        } catch (ClassCastException cce) {
            throw new InvalidKeyException("Wrong private key type");
        }
    }

    public void engineUpdate(byte b) throws SignatureException {
        try {
            md.update(b);
        } catch (NullPointerException npe) {
            throw new SignatureException("No SHA digest found");
        }
    }

    public void engineUpdate(byte b[], int offset, int length)
                                throws SignatureException {
        try {
            md.update(b, offset, length);
        } catch (NullPointerException npe) {
            throw new SignatureException("No SHA digest found");
        }
    }

    public byte[] engineSign(  ) throws SignatureException {
        byte b[] = null;
        try {
            b = md.digest(  );
        } catch (NullPointerException npe) {
            throw new SignatureException("No SHA digest found");
        }
        return crypt(b, priv);
    }

    public boolean engineVerify(byte[] sigBytes)
                                throws SignatureException {
        byte b[] = null;
        try {
            b = md.digest(  );
        } catch (NullPointerException npe) {
            throw new SignatureException("No SHA digest found");
        }
        byte sig[] = crypt(sigBytes, pub);
        return MessageDigest.isEqual(sig, b);
    }

    public void engineSetParameter(String param, Object value) {
        throw new InvalidParameterException("No parameters");
    }
    public void engineSetParameter(AlgorithmParameterSpec aps) {
        throw new InvalidParameterException("No parameters");
    }

    public Object engineGetParameter(String param) {
        throw new InvalidParameterException("No parameters");
    }

    public void engineReset(  ) {
    }

    private byte[] crypt(byte s[], DSAKey key) {
        DSAParams p = key.getParams(  );
        int rotValue = p.getP().intValue(  );
        byte d[] = rot(s, (byte) rotValue);
        return d;
    }

    private byte[] rot(byte in[], byte rotValue) {
        byte out[] = new byte[in.length];
        for (int i = 0; i < in.length; i++) {
            out[i] = (byte) (in[i] ^ rotValue);
        }
        return out;
    }
}
```

Like all implementations of engines in the security architecture, this class must have a constructor that takes no arguments, but it must call its superclass with its name. The constructor also is responsible for creating the instance of the underlying message digest using whatever algorithm this class feels is important. It is interesting to note that this requires the constructor to specify that it can throw a `NoSuchAlgorithmException` (in case the SHA algorithm can't be found).

The keys for this test algorithm are required to be DSA public and private keys. In general, the correspondence between an algorithm and the type of key it requires is very strong, so this is a typical usage. Hence, the two engine initialization methods cast the key to make sure that the key has the correct format. The engine initialization methods are not required to keep track of the state of the signature object −− that is, whether the object has been initialized for signing or for verifying. That logic, since it is common to all signature objects, is present in the generic initialization methods of the `Signature` class itself.

The methods that update the engine can simply pass their data to the message digest since the message digest is responsible for providing the fingerprint of the data that this object is going to sign or verify. Hence, the only interesting logic in this class is that employed by the signing and verification methods. Each method uses the message digest to create the digital fingerprint of the data. Then, to sign the data, the digest must be encrypted or otherwise operated upon with the previously defined private key −− this produces a unique digest that could only have been produced by the given data and the given private key. Conversely, to verify the data, the digest must be decrypted or otherwise operated upon with the previously defined public key; the resulting digest can then be compared to the expected digest to test for verification.

Clearly, the security of this algorithm depends on a strong implementation of the signing operations. Our example here does not meet that definition −− we're simply XORing every byte of the digest with a byte obtained from the parameters used to generate the keys. This XOR−encryption provides a good example since it's both simple and symmetric; a real digital signature implementation is much more complex.

These engine signing and verification methods are also responsible for setting the internal state of the engine back to an initialization state so that the same object can be used to sign or verify multiple signatures. In this case, no other work needs to be done for that; the message digest object itself is already reset once it creates its digest, and there is no other internal state inside the algorithm that needs to be reset. But if there were other state information, it would need to be reset in those methods.

12.4 Comparison with Previous Releases

There are few changes to the `Signature` class itself between Java 1.1 and Java 2. In Java 1.1, there is no `SignatureSpi` class and the `Signature` class extends the `Object` class instead; the `setParameter( )` method that requires an algorithm parameter spec does not exist in 1.1. In 1.1 and Java 2, version 1.2, the default security provider supports only DSA signatures; to get RSA signatures you must either install a third−party security provider or upgrade to 1.3. The `SignedObject` class is only available in Java 2.

There are significant changes to the way in which signed classes are handled between Java 1.1. and Java 2. In Java 1.1, there is no `jarsigner` tool; the equivalent tool is called `javakey`, and it creates signatures using the 1.1 identity scope (rather than a keystore). We will discuss this in Appendix C.

Since Java 1.1 does not have code sources, reading a signed jar file is also different. In fact, since the `java.util.jar` package does not exist in that release, the classes required to read a standard PKCS7 signature block are unavailable to us. More important, the security manager must handle signed classes differently: the class loader we presented here must be modified to associate the certificates with the class using the `setSigners( )` method of the `Class` class, and the security manager must retrieve those certificates with the `getSigners( )` method. In general, the security manager and the class loader must be more tightly−coupled in order for this all to work; that's a technique we'll show in Appendix D.

12.5 Summary

The digital signatures we've examined in this chapter form a key piece of the Java security architecture since they are the mechanism by which the parameters of the Java security sandbox can be extended: a digital signature gives the user the assurance that particular Java classes were provided by known entities. The user is then free to adopt a security policy for those classes based on the user's assessment of the trustworthiness of the entity that provided the classes. Digital signatures have many other uses, of course, and in conjunction with the `SignedObject` class they allow you to send and verify arbitrary pieces of data.

The digital signature engine is interesting also because it requires the use of the other engines we've looked at in earlier chapters −− the message digest engine to generate the fingerprint of the data that the digital signature will sign and the key pair engine (and its related classes) to provide the necessary keys to feed into this engine.

Chapter 13. Cipher−Based Encryption

In this chapter, we'll look at how to encrypt data using ciphers. We usually think of encryption as a means of protecting data sent over an insecure network, although it may also be used to protect data stored in a file, on a Java smart card, or in a number of other applications. With ciphers, the encryption of data is separate from its transmission. This is in sharp contrast to SSL, which can encrypt only data that is sent over sockets.

Cipher−based encryption is part of the JCE, which contains an engine (the cipher engine) that performs encryption as well as several classes that support data encryption. All the classes in this chapter are available only with the security provider that comes with JCE.

13.1 The Cipher Engine

First, we'll look at the engine that performs encryption within JCE. This engine is called the `Cipher` class (`javax.crypto.Cipher`); it provides an interface to encrypt and decrypt data either in arrays within the program or as that data is read or written through Java's stream interfaces:

public class Cipher implements Cloneable

Perform encryption and decryption of arbitrary data, using (potentially) a wide array of encryption algorithms.

Like all security engines, the cipher engine implements named algorithms. However, the naming convention for the cipher engine is different in that cipher algorithms are compound names that can include the name of the algorithm along with the name of a padding scheme and the name of a mode. Padding schemes and modes are specified by names −− just like algorithms. In theory, just as you may pick a new name for an algorithm, you may specify new names for a padding scheme or a mode, although the `SunJCE` security provider specifies several standard ones.

Modes and padding schemes are present in the `Cipher` class because that class implements what is known as a block cipher; that is, it expects to operate on data one block (e.g., 8 bytes) at a time. Padding schemes are required in order to ensure that the length of the data is an integral number of blocks.

Modes are provided to further alter the encrypted data in an attempt to make it harder to break the encryption. For example, if the data to be encrypted contains a number of similar patterns −− repeated names or header/footer information, for example −− any patterns in the resulting data may aid in breaking the encryption. Different modes of encrypting data help prevent these sorts of attacks. Depending upon the mode used by a cipher, it may need to be initialized in a special manner when the cipher is used for decryption. Some modes require initialization via an initialization vector.

Modes also enable a block cipher to behave as a stream cipher; that is, instead of requiring a large, 8−byte chunk of data to operate upon, a mode may allow data to be processed in smaller quantities. So modes are very important in stream−based operations where data may need to be transmitted one or two characters at a time.

The algorithms specified by the `SunJCE` security provider are:

DES

DES is the Data Encryption Standard algorithm, a standard that has been adopted by various organizations, including the U.S. government. There are known ways to attack this encryption, though they require a lot of computing power to do so; despite widespread predictions about the demise of

DES, it continues to be used in many applications and is generally considered secure. The examples in this chapter are mostly based on DES encryption.

DESede

This is also known as triple−DES or multiple−DES. This algorithm uses multiple DES keys to perform three rounds of DES encryption or decryption; the added complexity greatly increases the amount of time required to break the encryption. It also greatly increases the amount of time required to encrypt and to decrypt the data.

From a developer's perspective, DESede is equivalent to DES; only the algorithm name passed to the key generator and cipher engines is different. Although DESede requires multiple keys, these keys are encoded into a single secret key. Hence, the programming steps required to use DESede are identical to the steps required to use DES.

PBEWithMD5AndDES

This is the password−based encryption defined in PKCS#5. This algorithm entails using a password, a byte array known as salt, and an iteration count along with an MD5 message digest to produce a DES secret key; this key is then used to perform DES encryption or decryption. PKCS#5 was developed by RSA Data Security, Inc., primarily to encrypt private keys, although it may be used to encrypt any arbitrary data.

From a developer's perspective, this algorithm requires some special programming to obtain the key. We'll show an example of that later.

Blowfish

This algorithm was designed by Bruce Schneier; it is an attractive algorithm because it is not patented, and Mr. Schneier has placed implementations of the algorithm in the public domain. It is best used in applications where the key does not change often, though it requires a lot of memory.

The modes specified by the `SunJCE` security provider are:

ECB

This is the electronic cookbook mode. ECB is the simplest of all modes; it takes a simple block of data (8 bytes in the `SunJCE` implementation, which is standard) and encrypts the entire block at once. No attempt is made to hide patterns in the data, and the blocks may be rearranged without affecting decryption (though the resulting plaintext will be out of order). Because of these limitations, ECB is recommended only for binary data; text or other data with patterns in it is not well−suited for this mode.

ECB mode can only operate on full blocks of data, so it is generally used with a padding scheme.

ECB mode does not require an initialization vector.

CBC

This is the cipher block chaining mode. In this mode, input from one block of data is used to modify the encryption of the next block of data; this helps to hide patterns (although data that contains identical initial text −− such as email messages −− will still show an initial pattern). As a result, this mode is suitable for text data. This is the only mode that will work for PBEWithMD5AndDES encryption.

CBC mode can only operate on full blocks of data, so it is generally used with a padding scheme.

CBC mode requires an initialization vector for decryption.

CFB

This is the cipher−feedback mode. This mode is very similar to CBC, but its internal implementation is slightly different. CBC requires a full block (8 bytes) of data to begin its encryption, while CFB can begin encryption with a smaller amount of data. So this mode is suitable for encrypting text, especially when that text may need to be processed a character at a time. By default, CFB mode operates on 8−byte (64−bit) blocks, but you may append a number of bits after CFB (e.g., CFB8) to specify a different number of bits on which the mode should operate. This number must be a multiple of 8.

CFB requires that the data be padded so that it fills a complete block. Since that size may vary, the padding scheme that is used with it must vary as well. For CFB8, no padding is required, since data is always fed in an integral number of bytes.

CFB mode requires an initialization vector for decryption.

OFB

This is the output−feedback mode. This mode is also suitable for text; it is used most often when there is a possibility that bits of the encrypted data may be altered in transit (e.g., over a noisy modem). While a 1−bit error would cause an entire block of data to be lost in the other modes, it only causes a loss of 1 bit in this mode. By default, OFB mode operates on 8−byte (64−bit) blocks, but you may append a number of bits after OFB (e.g., OFB8) to specify a different number of bits on which the mode should operate. This number must be a multiple of 8.

OFB requires that the data be padded so that it fills a complete block. Since that size may vary, the padding scheme that is used with it must vary as well. For OFB8, no padding is required since data is always fed in an integral number of bytes.

OFB mode requires an initialization vector for decryption.

PCBC

This is the propagating cipher block chaining mode. It is popular in a particular system known as Kerberos; if you need to speak to a Kerberos version 4 system, this is the mode to use. However, this mode has some known methods of attack, and Kerberos version 5 has switched to using CBC mode. Hence, PCBC mode is no longer recommended.

PCBC mode requires that the input be padded to a multiple of 8 bytes.

PCBC mode requires an initialization vector for decryption.

The padding schemes specified by the `SunJCE` security provider are:

PKCS5Padding

This padding scheme ensures that the input data is padded to a multiple of 8 bytes.

NoPadding

When this scheme is specified, no padding of input is done. In this case, the number of input bytes presented to the encryption cipher must be a multiple of the block size of the cipher; otherwise, when the cipher attempts to encrypt or decrypt the data, it generates an error.

Remember these uses of modes and padding are specific to the `SunJCE` security provider. The modes and padding schemes are based upon accepted standards and are thus likely to be implemented in this manner by third−party security providers as well, but check your third−party provider documentation to be sure.

The mode and padding scheme specified for decryption must match the mode and padding scheme specified for encryption, or the decryption will fail.

The `Cipher` class is normally used to encrypt or decrypt data. However, it may also be used to wrap and unwrap keys. We'll discuss encryption and decryption first; information about wrapping and unwrapping keys will follow.

13.1.1 Using the Cipher Class for Encryption/Decryption

In order to obtain an instance of the `Cipher` class, we call one of these methods:

public static Cipher getInstance(String algorithmName)
public static Cipher getInstance(String algorithmName, String provider)

Obtain a cipher engine that can perform encryption and decryption by implementing the named algorithm. The engine is provided by the given security provider, or the list of installed security providers is searched for an appropriate engine.

If an implementation of the given algorithm cannot be found, a `NoSuchAlgorithmException` is thrown. If the named provider cannot be found, a `NoSuchProviderException` is thrown.

The algorithm name passed to the `getInstance( )` method may either be a simple algorithm name (e.g., DES), or it may be an algorithm name that specifies a mode and padding in this format: algorithm/mode/padding (e.g., DES/ECB/PKCS5Padding). If the mode and padding are not specified, they default to an implementation−specific value; in the `SunJCE` security provider, the mode defaults to ECB (CBC if the algorithm is PBEWithMD5andDES) and padding defaults to PKCS5.

Once you've obtained a cipher object, you must initialize it. An object can be initialized for encryption, decryption, or for key wrapping, but in any case, you must provide a key. If the algorithm is a symmetric cipher, you should provide a secret key; otherwise, you should provide a public key to encrypt data and a private key to decrypt data (in fact, the key must match the algorithm type: a DES cipher must use a DES key, and so on). Initialization is achieved with one of these methods:

public final void init(int op, Key k)
public final void init(int op, Certificate c)
public final void init(int op, Key k, AlgorithmParameterSpec aps)
public final void init(int op, Key k, AlgorithmParameterSpec aps, SecureRandom sr)
public final void init(int op, Key k, SecureRandom sr)

public final void init(int op, Certificate c, SecureRandom sr)
public final void init(int op, Key k, AlgorithmParameters ap)
public final void init(int op, Key k, AlgorithmParameters ap, SecureRandom sr)

Initialize the cipher to encrypt or decrypt data. If `op` is `Cipher.ENCRYPT_MODE`, the cipher is initialized to encrypt data; if `op` is `Cipher.DECRYPT_MODE`, the cipher is initialized to decrypt data. The cipher is initialized with the given key or the public key contained within the given certificate.

These calls reset the engine to an initial state, discarding any previous data that may have been fed to the engine. Hence, a single cipher object can be used to encrypt data and then later to decrypt data.

Many algorithm modes we discussed earlier require an initialization vector to be specified when the cipher is initialized for decrypting. In these cases, the initialization vector must be passed to the `init( )` method within the algorithm parameter specification or algorithm parameters; typically, the `IvParameterSpec` class is used to do this for DES encryption.

In the `SunJCE` security provider, specifying an initialization vector for a mode that does not support it will eventually lead to a `NullPointerException`. Failure to specify an initialization vector for a mode that requires one will generate incorrect decrypted data.

After an engine has been initialized, it must be fed data. There are two sets of methods to accomplish this. The first set can be used any number of times:

public final byte[] update(byte[] input)
public final byte[] update(byte[] input, int offset, int length)
public final int update(byte[] input, int offset, int length, byte[] output)
public final int update(byte[] input, int offset, int length, byte[] output, int outOffset)

Encrypt or decrypt the data in the input array (starting at the given offset for the given length, if applicable). The resulting data is either placed in the given output array (in which case the size of the output data is returned) or returned in a new array. If the cipher has not been initialized, an `IllegalStateException` is thrown.

If the length of the data passed to this method is not an integral number of blocks, any extra data is buffered internally within the cipher engine; the next call to an `update( )` or `doFinal( )` method processes that buffered data as well as any new data that is just being provided.

If the given output buffer is too small to hold the data, a `ShortBufferException` is thrown. The required size of the output buffer can be obtained from the `getOutputSize( )` method. A `ShortBufferException` does not clear the state of the cipher: any buffered data is still held, and the call can be repeated (with a correctly sized buffer) with no ill effects.

This second set of methods should only be called once:

public final byte[] doFinal( )
public final int doFinal(byte[] output, int offset)
public final byte[] doFinal(byte[] input)
public final byte[] doFinal(byte[] input, int offset, int length)
public final int doFinal(byte[] input, int offset, int length, byte[] output)
public final int doFinal(byte[] input, int offset, int length, byte[] output, int outOffset)

Encrypt or decrypt the data in the input array as well as any data that has been previously buffered in the cipher engine. This method behaves exactly the same as the `update( )` method, except that this method signals that all data has been fed to the engine. If the engine is performing padding, the padding scheme will be used to process the pad bytes (i.e., add padding bytes for encryption and remove padding bytes for decryption). If the cipher engine is not performing padding and the total of all processed data is not a multiple of the mode's block size, an `IllegalBlockSizeException` is thrown.

These methods throw an `IllegalStateException` or a `ShortBufferException` in the same circumstances as the `update( )` methods.

In order to initialize some ciphers for decryption, you need to specify an initialization vector; this initialization vector must be the same vector that was used when the cipher was initialized for encryption. For encryption, you may specify the initialization vector, or you may use a system−provided initialization vector. In order to retrieve this vector for later use (e.g., to send it to someone who will eventually need to decrypt the data), you may use this method:

public final byte[] getIV( )

Return the initialization vector that was used to initialize this cipher. If a system−provided initialization vector is used, that vector is not available until after the first call to an `update( )` or `doFinal( )` method.

In order to preallocate an output buffer for use in the `update( )` and `doFinal( )` methods, you must know its size, which is returned from this method:

public final int getOutputSize(int inputLength)

Return the output size for the next call to the `update( )` or `doFinal( )` methods, assuming that one of those methods is called with the specified amount of data. Note that the size returned from this call includes any possible padding that the `doFinal( )` method might add. A call to the `update( )` method may actually generate less data than this method would indicate because it will not create any padding.

Finally, there are two miscellaneous methods of this class:

public final Provider getProvider( )

Return the provider class that defined this engine.

public final int getBlockSize( )

Get the block size of the mode of the algorithm that this cipher implements.

Let's put this all together into a simple example:

```
package javasec.samples.ch13;

import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class CipherTest {
    public static void main(String args[]) {
        try {
            // First, we need a key; we'll just generate one
            // though we could look one up in the keystore or
            // obtain one via a key agreement algorithm.
            KeyGenerator kg = KeyGenerator.getInstance("DES");
            Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
            Key key = kg.generateKey(  );

            // Now we'll do the encryption. We'll also retrieve
            // the initialization vector, which the engine will
            // calculate for us.
            c.init(Cipher.ENCRYPT_MODE, key);
            byte input[] = "Stand and unfold yourself".getBytes(  );
            byte encrypted[] = c.doFinal(input);
            byte iv[] = c.getIV(  );

            // Now we'll do the decryption. The initialization
            // vector can be transmitted to the recipient with
            // the ciphertext, but the key must be transmitted
            // securely.
            IvParameterSpec dps = new IvParameterSpec(iv);
            c.init(Cipher.DECRYPT_MODE, key, dps);
            byte output[] = c.doFinal(encrypted);
            System.out.println("The string was ");
            System.out.println(new String(output));
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

We've reused the single engine object to perform both the encryption and the decryption. Since DES is a symmetric encryption algorithm, we generated a single key that is used for both operations. Within the `try` block, the second block of code performs the encryption:

1. We initialize the cipher engine for encrypting.
2. We pass the bytes we want to encrypt to the `doFinal( )` method. Of course, we might have had any number of calls to the `update( )` method preceding this call, with data in any arbitrary amounts. Since we've specified a padding scheme, we don't have to worry about the size of the data we pass to the `doFinal( )` method.
3. Finally, we save the initialization vector the system provided to perform the encryption. Note that this step would not be needed for ECB mode, which doesn't require an initialization vector.

Performing the decryption is similar:

1. First, we initialize the cipher engine for decrypting. In this case, however, we must provide an initialization vector to initialize the engine in order to get the correct results (again, this would be unnecessary for ECB mode).
2. Next, we pass the encrypted data to the `doFinal( )` method. Again, we might have had multiple calls to the `update( )` method first.

In typical usage, of course, encryption is done in one program and decryption is done in another program. In the example above, this would entail that the initialization vector and the encrypted data be transmitted to a receiver; this may be done via a socket or a file or any other convenient means. There is no security risk in transmitting the initialization vector, as it has the same properties as the rest of the encrypted data. So you could simply write out the byte array to the data sink followed by the data itself; the receiver would read the byte array, use it to construct the `IvParameterSpec` object, and then decrypt the data.

The NullCipher Class

The JCE includes one subclass of the `Cipher` class: the `NullCipher` class (`javax.crypto.NullCipher`). This class performs no encryption. Data passes through the null cipher unchanged, and no padding or blocking is performed (the `getBlockSize( )` method will return 1). Unlike a traditional cipher engine, instances of the `NullCipher` class must be constructed directly:

```
Cipher c = new NullCipher(  );
```

This class can be used to test the logic of your program without actually encrypting or decrypting the data.

13.1.2 Performing Your Own Padding

In this example, we used the PKCS5 padding scheme to provide the necessary padding. This is by far the simplest way. If you want to do your own padding −− if, for example, you're using a CFB32 mode for some reason −− you need to do something like this:

```
Cipher c = Cipher.getInstance("DES/CFB32/NoPadding");
c.init(Cipher.ENCRYPT_MODE, desKey);
int blockSize = c.getBlockSize(  );
byte b[] = "This string has an odd length".getBytes(  );
byte padded[] = new byte[b.length + blockSize −(b.length % blockSize)];
System.arraycopy(b, 0, padded, 0, b.length);
for (int i = 0; i < blockSize − (b.length % blockSize); i++)
    padded[b.length + i] = 0;
byte output[] = c.doFinal(padded);
```

The problem with this code is that when the data is decrypted, there is no indication of how many bytes should be discarded as padding. PKCS5 and other padding schemes solve this problem by encoding that information into the padding itself.

13.1.3 Initialization of a PBEWithMD5AndDES Cipher

As we mentioned earlier, a password−based cipher cannot be initialized without special data that is passed via the algorithm specification. This data is known as the salt and iteration count. Hence, a password−based cipher is initialized as follows:

```
String password = "Come you spirits that tend on mortal thoughts";
byte[] salt = { (byte) 0xc9, (byte) 0x36, (byte) 0x78, (byte) 0x99,
                (byte) 0x52, (byte) 0x3e, (byte) 0xea, (byte) 0xf2 };
PBEParameterSpec paramSpec = new PBEParameterSpec(salt, 20);
PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(  ));
SecretKeyFactory kf = SecretKeyFactory.getInstance("PBEWithMD5AndDES");
SecretKey key = kf.generateSecret(keySpec);
Cipher c = Cipher.getInstance("PBEWithMD5AndDES");
c.init(Cipher.ENCRYPT_MODE, key, paramSpec);
```

The rationale behind this system is that it allows the password to be shared verbally (or otherwise) between participants in the cipher; rather than coding the password as we've done above, the user would presumably enter the password. Since these types of passwords are often easy to guess (a string comparison of the above password against the collected works of Shakespeare would guess the password quite easily, despite its length), the iteration and salt provide a means to massage the password into something more secure. The salt itself should be random, and the higher the iteration count, the more expensive a brute−force attack against the key becomes (though it also takes longer to generate the key itself).

Of course, despite the presence of the salt and iteration, the password chosen in the method should not be easy to guess in the first place: it should contain special characters, not be known quotes from literature, and follow all the other usual rules that apply to selecting a password.

13.1.4 Using the Cipher Class for Key Wrapping

When you need to send a secret key between two parties, you may need to encrypt the key in order to protect it. This is often done because symmetric cipher algorithms are generally faster than asymmetric ones, but sharing public keys is generally much easier than sharing secret keys. So you can use an asymmetric cipher to encrypt a secret key that is sent between two parties; that way the only out−of−band data that must be transferred between the parties is a public key certificate, and you can still use a fast encryption algorithm.

Because this is a common operation, the `Cipher` class provides methods that wrap (encrypt) and unwrap (decrypt) keys. These operations could be accomplished by using the `update( )` and/or `doFinal( )` methods of the `Cipher` class directly, but the methods we're about to examine operate directly on `Key` objects rather than on arrays of bytes, so they are more convenient to use.

To wrap keys, you initialize the cipher object using any of the `init( )` methods we listed previously; you must specify `Cipher.WRAP_MODE` as the operation. To unwrap a key, the process is the same except you specify `Cipher.UNWRAP_MODE`. Then, instead of using the `update( )` and `doFinal( )` methods, you use one of these methods:

public final byte[] wrap(Key key)
: Wrap (encrypt) a key. The cipher must be initialized for `WRAP_MODE`.

public final Key unwrap(byte[] key, String algorithm, int type)
: Unwrap a key. To unwrap the key, you must have the key data, and you must know the algorithm used to generate the key (e.g., DES or RSA) and the type of the key. The type must be either `Cipher.SECRET_KEY`, `Cipher.PUBLIC_KEY`, or `Cipher.PRIVATE_KEY`.

Here's an example of wrapping a key. We still must have a key to initialize the cipher; we'll use the password−based key encryption for that, but you'd often use a public or private key. Then we can wrap the key, and in the second part of the example we can do the reverse operation:

```
package javasec.samples.ch13;

import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class WrapTest {
    public static void main(String[] args) throws Exception {
        // This is the key we want to transmit to another party.
        KeyGenerator kg = KeyGenerator.getInstance("DESede");
        Key sharedKey = kg.generateKey(  );

        // We'll transmit it with PBE encryption, which is good for
        // sending keys but not arbitrary data.
        String password =
               "Come you spirits that tend on mortal thoughts";
        byte[] salt = { (byte) 0xc9, (byte) 0x36, (byte) 0x78,
                        (byte) 0x99, (byte) 0x52, (byte) 0x3e,
                        (byte) 0xea, (byte) 0xf2 };
        PBEParameterSpec paramSpec = new PBEParameterSpec(salt, 20);
        PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(  ));
        SecretKeyFactory kf =
                    SecretKeyFactory.getInstance("PBEWithMD5AndDES");
        SecretKey passwordKey = kf.generateSecret(keySpec);
        Cipher c = Cipher.getInstance("PBEWithMD5AndDES");
        c.init(Cipher.WRAP_MODE, passwordKey, paramSpec);
        byte[] wrappedKey = c.wrap(sharedKey);

        // Now we'll initialize a cipher with the
        // DESede key and encrypt some data with it.
        c = Cipher.getInstance("DESede");
        c.init(Cipher.ENCRYPT_MODE, sharedKey);
        byte[] input = "Stand and unfold yourself".getBytes(  );
        byte[] encrypted = c.doFinal(input);

        // Now we'll try to read the wrapped key and the encrypted data
        // First, we have to unwrap the key.
        c = Cipher.getInstance("PBEWithMD5AndDES");

        // We'll reuse the key and param spec from before,
        // but generally we'd have to recreate it from the password.
        c.init(Cipher.UNWRAP_MODE, passwordKey, paramSpec);
        Key unwrappedKey = c.unwrap(wrappedKey, "DESede",
                                    Cipher.SECRET_KEY);

        // Now we can use the unwrapped key to decrypt the data.
        c = Cipher.getInstance("DESede");
        c.init(Cipher.DECRYPT_MODE, unwrappedKey);
        String newData = new String(c.doFinal(encrypted));
        System.out.println("The string was " + newData);
    }
}
```

13.1.5 Implementing the Cipher Class

As with all Java 2 engines, the SPI for the `Cipher` class is a separate class, the `CipherSpi` class (`javax.crypto.CipherSpi`):

public abstract class CipherSpi

The SPI for the `Cipher` class. This class is responsible for performing the encryption or decryption according to its internal algorithm. Support for various modes or padding schemes must be handled by this class as well.

There is very little intelligence in the `Cipher` class itself; virtually all of its methods are simply passed through calls to corresponding methods in the SPI. The one exception to this is the `getInstance( )` method, which is responsible for parsing the algorithm string and removing the mode and padding strings if present. If it finds a mode and padding specification, it calls these methods of the SPI:

public abstract void engineSetMode(String s)

Set the mode of the cipher engine according to the specified string. If the given mode is not supported by this cipher, a `NoSuchAlgorithmException` should be thrown.

public abstract void engineSetPadding(String s)

Set the padding scheme of the cipher engine according to the specified string. If the given padding scheme is not supported by this cipher, a `NoSuchPaddingException` should be thrown.

Remember that the mode and padding strings we looked at earlier are specific to the implementation of the `SunJCE` security provider. Hence, while ECB is a common mode specification, it is completely at the discretion of your implementation whether that string should be recognized or not. If you choose to implement a common mode, it is recommended that you use the standard strings, but you may use any naming convention that you find attractive. The same is true of padding schemes.

Complicating this matter is the fact that there are no classes in JCE that assist you with implementing any mode or padding scheme. So if you need to support a mode or padding scheme, you must write the required code from scratch.

The remaining methods of the SPI are all called directly from the corresponding methods of the `Cipher` class:

protected abstract int engineGetBlockSize( )

Return the number of bytes that comprise a block for this engine. Unless the cipher is capable of performing padding, input data for this engine must total a multiple of this block size (though individual calls to the `update( )` method do not necessarily have to provide data in block−sized chunks).

protected abstract byte[] engineGetIV( )

Return the initialization vector that was used to initialize the cipher. If the cipher was in a mode where no initialization vector was required, this method should return `null`.

protected abstract int engineGetOutputSize(int inputSize)

Return the number of bytes that the cipher will produce if the given amount of data is fed to the cipher. This method should take into account any data that is presently being buffered by the cipher as well as any padding that may need to be added if the cipher is performing padding.

protected void engineInit(int op, Key key, SecureRandom sr)
protected void engineInit(int op, Key key, AlgorithmParameterSpec aps, SecureRandom sr)
protected void engineInit(int op, Key key, AlgorithmParameters ap, SecureRandom sr)

Initialize the cipher based on the `op`, which will be either `Cipher.ENCRYPT_MODE`, `Cipher.DECRYPT_MODE`, `Cipher.WRAP_MODE`, or `Cipher.UNWRAP_MODE`. This method should ensure that the key is of the correct type and throw an `InvalidKeyException` if it is not (or if it is otherwise invalid), and use the given random number generator (and algorithm parameters, if applicable) to initialize its internal state. If algorithm parameters are provided but not supported or are otherwise invalid, this method should throw an `InvalidAlgorithmParameterException`.

protected abstract byte[] engineUpdate(int input[], int offset, int len)
protected abstract int engineUpdate(int input[], int offset, int len, byte[] output, int outOff)

Encrypt or decrypt the input data. The data that is passed to these methods is not necessarily an integral number of blocks. It is the responsibility of these methods to process as much of the input data as possible and to buffer the remaining data internally. Upon the next call to an `engineUpdate( )` or `engineDoFinal( )` method, this buffered data must be processed first, followed by the input data of that method (and again leaving any leftover data in an internal buffer).

protected abstract byte[] engineDoFinal(int input[], int offset, int len)
protected abstract int engineDoFinal(int input[], int offset, int len, byte[] output, int outOff)

Encrypt or decrypt the input data. Like the `update( )` method, this method must consume any buffered data before processing the input data. However, since this is the final set of data to be processed, this method must make sure that the total amount of data has been an integral number of blocks; it should not leave any data in its internal buffers.

If the cipher supports padding (and padding was requested through the `engineSetPadding( )` method), this method should perform the required padding; an error in padding should cause a `BadPaddingException` to be thrown. Otherwise, if padding is not being performed and the total amount of data has not been an integral number of blocks, this method should throw an `IllegalBlockSizeException`.

protected byte[] engineWrap(Key key)

Wrap the key, returning the encrypted bytes that can be used to reconstitute the key. Note that this method is not abstract (for backward−compatibility reasons). You must override it if you want to support key wrapping; otherwise it will throw an `UnsupportedOperationException`. If the key is not the correct format, you should throw an `InvalidKeyException`; if the cipher cannot handle the correct block size of the key then it should throw an `IllegalBlockSizeException`.

protected Key engineUnwrap(byte[] key, String algorithm, int type)

Unwrap the key bytes, returning the reconstituted key. Note that this method is not abstract (for backward−compatibility reasons); by default it throws an `UnsupportedOperationException`. It should throw a `NoSuchAlgorithmException` if the algorithm type isn't supported, or an `InvalidKeyException` if the key can't be unwrapped.

Using our typical XOR strategy of encryption, here's a simple implementation of a cipher engine:

```
package javasec.samples.ch13;

import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javasec.samples.ch08.XYZProvider;
import javasec.samples.ch09.XORKey;

public class XORCipher extends CipherSpi {
    byte xorByte;

    public XORCipher(  ) {
        XYZProvider.verifyForJCE(  );
    }

    protected void engineInit(int i, Key k, SecureRandom sr)
                        throws InvalidKeyException {
        if (!(k instanceof XORKey))
            throw new InvalidKeyException("XOR requires an XOR key");
        xorByte = k.getEncoded(  )[0];
    }
    protected void engineInit(int i, Key k, AlgorithmParameterSpec aps,
                    SecureRandom sr) throws InvalidKeyException,
                            InvalidAlgorithmParameterException {
        throw new InvalidAlgorithmParameterException(
            "Algorithm parameters not supported in this class");
    }

    protected void engineInit(int i, Key k, AlgorithmParameters ap,
            SecureRandom sr) throws InvalidKeyException,
            InvalidAlgorithmParameterException {
        throw new InvalidAlgorithmParameterException(
            "Algorithm parameters not supported in this class");
    }

    protected byte[] engineUpdate(byte in[], int off, int len) {
        return engineDoFinal(in, off, len);
    }

    protected int engineUpdate(byte in[], int inoff, int length,
                            byte out[], int outoff) {
        for (int i = 0; i < length; i++)
            out[outoff + i] = (byte) (in[inoff + i] ^ xorByte);
        return length;
    }

    protected byte[] engineDoFinal(byte in[], int off, int len) {
        byte out[] = new byte[len − off];
        engineUpdate(in, off, len, out, 0);
        return out;
    }

    protected int engineDoFinal(byte in[], int inoff, int len,
                                byte out[], int outoff) {
        return engineUpdate(in, inoff, len, out, outoff);
    }

    protected int engineGetBlockSize(  ) {
        return 1;
    }

    protected byte[] engineGetIV(  ) {
        return null;
    }

    protected int engineGetOutputSize(int sz) {
        return sz;
    }

    protected void engineSetMode(String s)
                        throws NoSuchAlgorithmException {
        throw new NoSuchAlgorithmException("Unsupported mode " + s);
    }

    protected void engineSetPadding(String s)
                        throws NoSuchPaddingException {
        throw new NoSuchPaddingException("Unsupported padding " + s);
    }

    protected AlgorithmParameters engineGetParameters(  ) {
        return null;
    }
}
```

The bulk of the work of any cipher engine will be in the `engineUpdate( )` method, which is responsible for actually providing the ciphertext or plaintext. In this case, we've simply XORed the key value with every byte, a process that works both for encryption as well as decryption. Because the work done by the `engineUpdate( )` method is so symmetrical, we don't need to keep track internally of whether we're encrypting or decrypting; for us, the work is always the same. For some algorithms, you may need to keep track of the state of the cipher by setting an internal variable when the `engineInit( )` method is called.

Similarly, because we can operate on individual bytes at a time, we didn't have to worry about padding and buffering internal data. Such an extension is easy using the code we showed earlier that uses the modulus operator to group the input arrays into blocks.

To use this class, we would need to use the `XYZProvider` class we developed in Chapter 8. Then we simply instantiate the engine and create the cipher like this:

```
Security.addProvider(new XYZProvider(  ));
KeyGenerator kg = KeyGenerator.getInstance("XOR");
Cipher c = Cipher.getInstance("XOR");
```

Note that "XOR" is the valid algorithm name for this implementation since we do not support any modes or padding schemes. Note too that we no longer need an initialization vector to create the cipher. Finally, remember that the `Cipher` class is a JCE engine, which is why the constructor here calls the `verifyForJCE( )` method.

13.2 Cipher Streams

In the `Cipher` class we just examined, we had to provide the data to be encrypted or decrypted as multiple blocks of data. This is not necessarily the best interface for programmers: what if we want to send and receive arbitrary streams of data over the network? It would often be inconvenient to get all the data into buffers before it can be encrypted or decrypted.

The solution to this problem is the ability to associate a cipher object with an input or output stream. When data is written to such an output stream, it is automatically encrypted, and when data is read from such an input stream, it is automatically decrypted. This allows a developer to use Java's normal semantics of nested filter streams to send and receive encrypted data.

13.2.1 The CipherOutputStream Class

The class that encrypts data on output to a stream is the `CipherOutputStream` class (`javax.crypto.CipherOutputStream`):

public class CipherOutputStream extends FilterOutputStream

Provide a class that will encrypt data as it is written to the underlying stream.

Like all classes that extend the `FilterOutputStream` class, constructing a cipher output stream requires that an existing output stream has already been created. This allows us to use the existing output stream from a socket or a file as the destination stream for the encrypted data:

public CipherOutputStream(OutputStream outputStream, Cipher cipher)

Create a cipher output stream, associating the given cipher object with the existing output stream. The given cipher must already have been initialized, or an `IllegalStateException` will be thrown.

The output stream may be operated on with any of the methods from the `FilterOutputStream` class --

the `write( )` methods, the `available( )` method, and the `close( )` method, which all provide the semantics you would expect. Often, of course, these methods are never used directly −− for example, if you're sending text data over a socket, you will wrap a cipher output stream around the socket's output stream, but then you will wrap a print writer around that; the programming interface then becomes a series of calls to the `print( )` and `println( )` methods. You can use any similar output stream to get a different interface.

It does not matter if the cipher object that was passed to the constructor does automatic padding or not −− the `CipherOutputStream` class itself does not make that restriction. As a practical matter, however, you'll want to use a padding cipher object, since otherwise you'll be responsible for keeping track of the amount of data passed to the output stream and tacking on your own padding.

Usually, the better alternative is to use a byte−oriented mode such as CFB8. This is particularly true in streams that are going to be used conversationally: a message is sent, a response received, and then another message is sent, etc. In this case, you want to make sure that the entire message is sent; you cannot allow the cipher to buffer any data internally while it waits for a full block to arrive. And, for reasons we're just about to describe, you cannot call the `flush( )` method in this case either. Hence, you need to use a streaming cipher (or, technically, a block cipher in streaming mode) in this case.

When the `flush( )` method is called on a `CipherOutputStream` (either directly or because the stream is being closed), the padding of the stream comes into play. If the cipher is automatically padding, the padding bytes are generated in the `flush( )` method. If the cipher is not automatically padding and the number of bytes that have been sent through the stream is not a multiple of the cipher's block size, then the `flush( )` method (or indirectly the `close( )` method) throws an `IllegalBlockSizeException` (note that this requires that the `IllegalBlockSizeException` be a runtime exception).

If the cipher is performing padding, it is very important not to call the `flush( )` method unless it is immediately followed by a call to the `close( )` method. If the `flush( )` method is called in the middle of processing data, padding is added in the middle of the data. This means the data does not decrypt correctly. Remember that certain output streams (especially some types of `PrintWriter` streams) flush automatically; if you're using a padding cipher, don't use one of those output streams.

We can use this class to write some encrypted data to a file like this:

```
package javasec.samples.ch13;

import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class Send {
    public static void main(String args[]) {
        try {
            KeyGenerator kg = KeyGenerator.getInstance("DES");
            kg.init(new SecureRandom(  ));
            SecretKey key = kg.generateKey(  );
            SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
            Class spec = Class.forName("javax.crypto.spec.DESKeySpec");
            DESKeySpec ks = (DESKeySpec) skf.getKeySpec(key, spec);
            ObjectOutputStream oos = new ObjectOutputStream(
                        new FileOutputStream("keyfile"));
            oos.writeObject(ks.getKey(  ));

            Cipher c = Cipher.getInstance("DES/CFB8/NoPadding");
            c.init(Cipher.ENCRYPT_MODE, key);
            CipherOutputStream cos = new CipherOutputStream(
                        new FileOutputStream("ciphertext"), c);
            PrintWriter pw = new PrintWriter(
                        new OutputStreamWriter(cos));
            pw.println("Stand and unfold yourself");
            pw.close(  );
            oos.writeObject(c.getIV(  ));
            oos.close(  );
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

There are two steps involved here. First, we must create the cipher object, which means that we must have a secret key available. Bypassing the secret key management examples we've used before, we're just going to save the key object to a file that can later be read by whomever needs the key. Note that we've gone through the usual steps of writing the data produced by the secret key factory so that the recipient of the key need not use the same provider we use.

After we generate the key, we must create the cipher object, initialize it with that key, and then use that cipher object to construct our output stream. Once the data is sent to the stream, we close the stream, which flushes the cipher object, performs any necessary padding, and completes the encryption.

In this case, we've chosen to use CFB8 mode, so there is no need for padding. But in general, this last step is important: if we don't explicitly close the `PrintWriter` stream, when the program exits, data that is buffered in the cipher object itself will not get flushed to the file. The resulting encrypted file will be unreadable, as it won't have the correct amount of data in its last block.[1]

[1] Closing the output stream is necessary whenever the stream performs buffering, but it is particularly important to remember in this context.

13.2.2 The CipherInputStream Class

The output stream is only half the battle; in order to read that data, we must use the `CipherInputStream` class (`javax.crypto.CipherInputStream`):

public class CipherInputStream extends FilterInputStream
: Create a filter stream capable of decrypting data as it is read from the underlying input stream.

A cipher input stream is constructed with this method:

public CipherInputStream(InputStream is, Cipher c)
: Create a cipher input stream that associates the existing input stream with the given cipher. The cipher must previously have been initialized.

All the points we made about the `CipherOutputStream` class are equally valid for the `CipherInputStream` class. You can operate on it with any of the methods in its superclass, although you'll typically want to wrap it in something like a buffered reader, and the cipher object that is associated with the input stream needs to perform automatic padding or use a mode that does not require padding (in fact, it must use the same padding scheme and mode that the output stream that is sending it data used).

The `CipherInputStream` class does not directly support the notion of a mark. The `markSupported( )` method returns `false` unless you've wrapped the cipher input stream around another class that supports a mark.

Here's how we could read the data file that we created above:

```
package javasec.samples.ch13;

import java.io.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

public class Receive {
    public static void main(String args[]) {
        try {
            ObjectInputStream ois = new ObjectInputStream(
                        new FileInputStream("keyfile"));
            DESKeySpec ks = new DESKeySpec((byte[]) ois.readObject(  ));
            SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
            SecretKey key = skf.generateSecret(ks);

            Cipher c = Cipher.getInstance("DES/CFB8/NoPadding");
            c.init(Cipher.DECRYPT_MODE, key,
                   new IvParameterSpec((byte[]) ois.readObject(  )));
            CipherInputStream cis = new CipherInputStream(
                        new FileInputStream("ciphertext"), c);
            BufferedReader br = new BufferedReader(
                        new InputStreamReader(cis));
            System.out.println("Got message");
            System.out.println(br.readLine(  ));
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

In this case, we must first read the secret key from the file where it was saved, then create the cipher object initialized with that key. Then we can create our input stream and read the data from the stream, automatically decrypting it as it goes.

13.3 Sealed Objects

The final class in JCE that we'll investigate is the `SealedObject` class (`javax.crypto.SealedObject`). This class is very similar to the `SignedObject` class we examined in Chapter 12, except that the stored, serialized object is encrypted rather than signed:

public class SealedObject

A class that can embed within it a serializable object in an encrypted form.

You can construct a sealed object as follows:

public SealedObject(Serializable obj, Cipher c)

Construct a sealed object. The sealed object serializes the given object to an embedded byte array, effectively making a copy of the object. It then uses the given cipher to encrypt the embedded byte array. If the object is unable to be serialized, an `IOException` is thrown; an error in encrypting the byte array results in an `IllegalBlockSizeException`. If the cipher object has not been initialized, an `IllegalStateException` is generated.

To retrieve the object, we use this method:

public Object getObject(Cipher c)

Decrypt the embedded byte array and deserialize it, returning the reconstituted object. The cipher must have been initialized with the same mode and key as the cipher that was passed to the constructor when the object was first created, otherwise a `BadPaddingMethodException` or an `IllegalBlockSizeException` is thrown. If the cipher was not initialized, an `IllegalStateException` is generated; failure to find the serialized class results in a `ClassNotFoundException`, and generic deserialization errors result in an `IOException`.

These are the only two operations that may be performed upon a sealed object. Keep in mind that the embedded object in this class is a serialized instance of the original object: the technique the object uses to perform serialization may affect the resulting object that is retrieved from the sealed object. This class can help us prevent someone from tampering with our serialized object, but the reconstituted object may be lacking transient fields or other information (depending, of course, on the implementation of the object itself).

13.4 Comparison with Previous Releases

The classes discussed in this chapter are available in version 1.2.1 of JCE. Version 1.2.1 is the first exportable release of JCE; previous versions may be used only within the United States and Canada. However, there are third−party implementations of earlier versions of JCE that were written by entities outside of the United States, and those implementations may be used outside of the United States.

Version 1.2 does not support key wrapping. The `init( )` methods to the `Cipher` class that require a certificate are not available in 1.2.

Version 1.1 of JCE supports only the DES, DESede, and PBEWithMD5AndDES encryption algorithms.

13.5 Summary

In this chapter, we explored cipher−based encryption engines. These engines perform encryption of arbitrary chunks or streams of data according to various algorithms. Cipher−based encryption engines separate the encryption of data from its transmission, so they are suitable for many purposes. They can be used to encrypt data to be sent over a socket, but they may also be used to encrypt data to be stored in a file, on a Java smart card, or even held in memory on an insecure machine.

In the next chapter, we'll look at another method of performing encryption: SSL, the secure sockets layer protocol, which automatically encrypts data sent over network sockets.

Chapter 14. SSL and HTTPS

In this chapter, we will explore how the Java Secure Socket Extension (JSSE) can be used to perform Secure Sockets Layer (SSL) encryption. SSL provides data encryption over TCP sockets and is the basis of the HTTPS protocol. JSSE provides a means to create and use SSL sockets as well as protocol handlers to support the HTTPS protocol. Because it is tightly coupled with TCP sockets, SSL is not a general−purpose encryption engine, but it is probably the most popular form of encryption used on the Internet. Its design has many advantages for network−based encryption as well.

14.1 An Overview of SSL and JSSE

On the Internet, data encryption is often performed using the Secure Sockets Layer protocol. This protocol was originally designed by Netscape for use in its browsers and secure servers; its most popular implementation is SSL 3.0. As SSL became a key technology on the Internet, its development and maintenance were taken over by the Internet Engineering Task Force (IETF), which made some slight modifications to SSL 3.0 and established the Transport Layer Security (TLS) Internet standard (presently version 1.0). Don't be confused by the names here: SSL and TLS are essentially the same protocol; TLS 1.0 is really just SSL version 3.1 (which has very few changes from version 3.0). TLS 1.0 is backward−compatible with SSL 3.0, so a program that uses the TLS protocol will be able to talk to other programs that use SSL 3.0.

SSL 3.0 is used by most browsers, including Netscape 4.x and 6, Internet Explorer 4.x and 5.x, the AOL browser, and the Opera browser. Some earlier versions of these browsers used SSL 2.0. SSL servers almost universally use 3.0 or later, partly because they are expected to talk to the newest browsers and partly because SSL 3.0 is considered more secure than SSL 2.0.

JSSE defines an API for SSL sockets. Sun provides a default implementation of that API in the reference implementation of JSSE. In theory, you can obtain third−party implementations of JSSE that plug into the JSSE framework, although as we'll see later in this chapter, you must use Sun's implementation−specific classes to perform certain operations. Hence, programs that use JSSE are not always implementation−independent. Sun's reference implementation of JSSE supports only SSL 3.0 and TLS 1.0. See Chapter 1, for details on how to obtain and install JSSE.

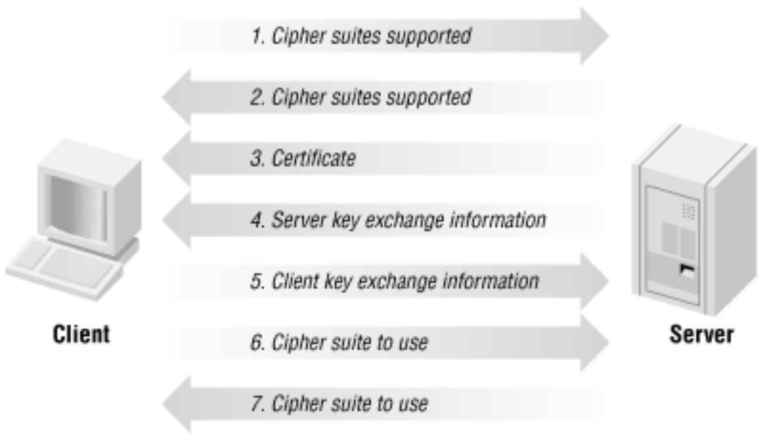
SSL, as its name implies, is designed to be used over sockets; there is no way within its protocol to separate the encryption from the data transmission. If you use SSL, you must use TCP sockets to transmit your data; UDP sockets and other transmission modes are not supported. Despite this, there are three key advantages to using SSL:

- It is ubiquitous. There are a number of existing services built on top of SSL (such as HTTPS), and to communicate with or implement such services, you must use SSL.
- It provides a simple interface to secret key encryption. Essentially, SSL performs two operations: a secret key exchange (like the Diffie−Hellman exchange we looked at in Chapter 10) and data encryption with the exchanged key. SSL hides most of the details of the key exchange and encryption, allowing developers to focus on their applications' logic.
- It is designed for an environment in which there are (relatively) few servers and many, many clients. SSL servers must provide credentials (i.e., a certificate) to authenticate themselves to their clients, but the clients do not need to provide credentials to the server (unless the server requires them to do so). Hence, the client in an SSL conversation does not need to go to the trouble of obtaining a certificate. This allows, for example, an Internet shopper to make purchases without obtaining a certificate.

  This makes sense on two levels. First, given the complexity of distributing certificates to thousands of clients, it prevents what might otherwise be a major roadblock in consumer acceptance of e−commerce. Second, the client already has other credentials that it gives to the server, such as a credit card number, an address, a promise to send a check, and so on. It's easier for the client to provide these credentials than to obtain a digital certificate in order to operate with an SSL server.

The SSL protocol exchanges several pieces of information between the client and server when the client first connects to the server. This process, known as an *SSL handshake* , is illustrated in Figure 14−1.

Figure 14−1. Sample SSL handshake


1. Cipher suites supported
2. Cipher suites supported
3. Certificate
4. Server key exchange information
5. Client key exchange information
6. Cipher suite to use
7. Cipher suite to use
Client
Server

Although there are variations on the way this proceeds (and there may be intermediate messages in the protocol), the basic flow of information goes like this:

1. The client initiates the connection to the server and tells the server which SSL cipher suites the client supports.
2. The server responds with the cipher suites that it supports.
3. The server sends the client a certificate that verifies its identity.
4. The server initiates a key exchange algorithm, based in part on the information contained in the certificate it has just sent, and sends the necessary key exchange information to the client.
5. The client completes the key exchange algorithm and sends the necessary key exchange information to the server. Along the way, it verifies the certificate.
6. Based on the type of key exchange algorithm (which in turn is based on the type of key in the server's certificate), the client selects an appropriate cipher suite and tells the server which suite it wants to use.
7. The server makes a final decision as to which cipher suite to use.

If the server desires, it can ask for the client's certificate so that it can be assured of who it is talking to; the client certificate is sent before the client key exchange information (if required). Once all this information has been established, the client and server can communicate normally over the socket; data that flows over the socket will be encrypted automatically.

Programatically, these steps are transparent: a developer asks for an SSL socket, and when the SSL socket is created, the protocol handshake is complete. This is one of the advantages of using SSL. We could have written all the code to establish the connection, perform the key exchange (using code based on the key exchange example we saw in Chapter 10), and encrypt the data (using code based on the examples from Chapter 13), but SSL hides all those details from us.

14.1.1 Keystores and Truststores

JSSE introduces the notion of a *truststore*, which is a database that holds certificates. In fact, a truststore has exactly the same format as a keystore; both are administered with `keytool`, and both are represented programmatically as instances of the `KeyStore` class. The difference between a keystore and a truststore is more a matter of function than of programming construct, as we will see.

The server in an SSL conversation must have a private key and a certificate that verifies its identity. The private key is used by the server as part of the key exchange algorithm, and the certificate is sent to the client to tell the client who the server is. This information is obtained from the keystore. Remember from our key exchange example in Chapter 10 that the private key is never sent from the server to the client; it is used only as an input to the key exchange algorithm.

SSL servers can require that the client authenticate itself as well. In that case, the client must have its own keystore with a private key and certificate.

The truststore is used by the client to verify the certificate that is sent by the server. If I set up an SSL server, it will use a certificate from my keystore to vouch for my identity. This certificate is signed by a trusted certificate authority (or, as we've seen, there may be a chain of certificates, the last of which is signed by a recognized CA). When your SSL client receives my certificate, it must verify that certificate, which means that the trusted CA's certificate must be in your local truststore. In general, all SSL clients must have a truststore. If an SSL server requires client authentication, it must also have a truststore.

In sum, keystores are used to provide credentials, while truststores are used to verify credentials. Servers use keystores to obtain the certificates they present to the clients; clients use truststores to obtain root certificates in order to verify the servers' certificates.

The keystore and truststore can be (and often are) the same file. However, it's usually easier to manage keys if they are separate: the truststore can contain the public certificates of trusted CAs and can be shared easily, while the keystore can contain the private key and certificate of the local organization and can be stored in a protected location. In addition, JSSE is easier to use if the keystore contains a single alias. When the keystore contains multiple aliases there are ways to specify which one should be used, but that requires more programming. We'll show that programming later in this chapter, but keep in mind that in general a keystore containing a single alias makes using JSSE simpler.

Keystores and truststores are managed programmatically by classes that implement the `KeyManager` and `TrustManager` interfaces. We'll look into those interfaces later, but for now we'll use their default implementations. The default trust manager assumes the public certificates are held in *$JREHOME/lib/security/jssecacerts*. If that file doesn't exist, it assumes they are held in *$JREHOME/lib/security/cacerts*. As we've seen, this file contains the root certificates of several well−known CAs, so it is a good choice. If you want to provide a different truststore, you can set the system property `javax.net.ssl.trustStore` to the name of the desired file (note that this property is not a URL, so you can only load it from a local file).

The default implementation of the key manager does not use a default keystore; it has no notion of *$HOME/.keystore*. To use the default key manager, you must set the `javax.net.ssl.keyStore` property to the name of the desired keystore. In addition, you must set the `javax.net.ssl.keyStorePassword` to the password for the desired keystore. The password in this case must be the same for both the keystore itself and the private key entry that you want to use.

For the most part SSL uses RSA keys, and those are the type of keys and certificates you need in your keystore and truststore. This is not an absolute requirement; SSL is actually very flexible about the types of key exchange and encryption algorithms it supports, and if you communicate only between JSSE servers and clients, you can use DSA keys. However, most browsers understand only RSA−based algorithms, so if you're writing a server that a browser will talk to, you need an RSA key.

In the examples in Chapter 10, we created a keystore with an RSA key. We'll use that keystore for the examples in this chapter, and we'll export the certificate from that keystore to use as the truststore. Here are the steps to do that:

1. Create an RSA key for the keystore. See Chapter 10 for information on the `genkey` option to `keytool` to do this. You can get a CA−issued certificate for this key if you want, but the examples in this chapter will work with the default (self−signed) certificate.
2. Export the RSA certificate (without the private key):

   ```
   piccolo% keytool −export −alias sdo −file test.cer
   Enter keystore password:  ******
   Certificate stored in file <test.cer>
   ```
3. Import the RSA certificate into a new file (the truststore). This allows us to recognize the issuer of the certificate as a valid CA. If you got a CA−issued certificate, you can use the `trustcacerts` option to accept the certificate automatically. We're storing the truststore in *$HOME/.truststore* ; on Microsoft Windows, you might substitute *C:\WINDOWS\ .truststore* (or any other file you like). Here's how to import the certificate:

   ```
   piccolo% keytool −import −alias test −keystore $HOME/.truststore −file test.cer
   Enter keystore password:  ******
   Owner: CN=Test Certificate, OU=My Test Organization, O="Me, Inc.", L=NY, ST=NY, C=US
   Issuer: CN=Test Certificate, OU=My Test Organization, O="Me, Inc.", L=NY, ST=NY, C=US
   Serial number: 39f3a2f3
   Valid from: Sun Oct 22 22:31:15 EDT 2000 until: Sat Jan 20 21:31:15 EST 2001
   Certificate fingerprints:
            MD5:  5E:B0:1C:D5:F6:2E:36:BF:F8:00:AA:4B:66:28:DE:DD
            SHA1: 28:B7:83:D2:0E:95:1D:EE:C3:D7:A9:D4:D5:1E:0E:82:E0:E9:F3:8D
   Trust this certificate? [no]:  yes
   Certificate was added to keystore
   ```

At this point, we have two files. The keystore file contains the private key and a certificate that vouches for our identity; we'll use that as the server's keystore. The truststore file contains just the certificate; we'll use it as the client's truststore.

14.1.2 JSSE Certificates

JSSE defines yet another certificate class, `javax.security.cert.Certificate`. This class is not related either to the `java.security.cert.Certificate` class that we've used in all our previous examples or to the deprecated `java.security.Certificate` interface. Since JSSE is designed to run on Personal Java implementations, for which the `java.security` package is optional, it cannot rely on classes from that package and must define its own certificate class.

Sun's reference implementation of JSSE, however, uses the `java.security.cert.Certificate` class internally, so it cannot run on versions of PersonalJava that do not supply the optional `java.security` package. This does not prevent third parties from implementing JSSE independent of the `java.security` package and providing those implementations with more limited PersonalJava packages. It does, however, limit the environments in which you can use Sun's JSSE implementation.

The APIs of the `java.security.cert.Certificate` and `javax.security.cert.Certificate` classes are identical. However, there are no facilities to convert easily between the two classes; you must get the encoded bytes from the javax.security.cert.Certificate and feed them through a java.security.cert.CertificateFactory object.

In addition, JSSE defines a javax.security.cert.X509Certificate class that is identical to the java.security.cert.X509Certificate class, although again it is unrelated to that class in the Java class hierarchy.

Handling JSSE certificates often requires that you parse the distinguished name (DN) held in the certificate. To make that easier, we'll use this class in our examples:

```
package javasec.samples.ch14;

// Store an X500 Name and extract its components on demand
public class X500Name {

    private String CN, OU, O, L, ST, C;
    private String name;
    private char nameChar[];

    public X500Name(String s) {
        if (s == null)
            throw new IllegalArgumentException("Name can't be null");
        name = s;
    }

    public String getCN(  ) {
        if (CN == null)
            CN = parse("CN=");
        return CN;
    }

    public String getOU(  ) {
        if (OU == null)
            OU = parse("OU=");
        return OU;
    }

    public String getO(  ) {
        if (O == null)
            O = parse("O=");
        return O;
    }

    public String getL(  ) {
        if (L == null)
            L = parse("L=");
        return L;
    }

    public String getST(  ) {
        if (ST == null)
            ST = parse("ST=");
        return ST;
    }

    public String getC(  ) {
        if (C == null)
            C = parse("C=");
        return C;
    }
    // Parse the name for the given target
    private String parse(String target) {
        if (nameChar == null)
            nameChar = name.toCharArray(  );
        char targetChar[] = target.toCharArray(  );

        for (int i = 0; i < nameChar.length; i++) {
            if (nameChar[i] == targetChar[0]) {
                // Possible match, check further
                boolean found = true;   // At least so far...
                for (int j = 0; j < targetChar.length; j++) {
                    try {
                        if (nameChar[i + j] != targetChar[j]) {
                            // No match, continue on...
                            found = false;
                            break;
                        }
                    } catch (ArrayIndexOutOfBoundsException aioobe) {
                        // No match, and nothing left in nameChar
                        return null;
                    }
                }
                if (found) {
                    int firstPos = i + targetChar.length;
                    int lastPos;
                    int endChar;
                    if (nameChar[firstPos] == '\"')
                        endChar = '\"';
                    else endChar = ',';
                    // The substring will be terminated by a quote if
                    // the substring is quoted (CN="My Name",OU=...)
                    // or by a comma otherwise (L=New York,ST=...)
                    // or by the end of the string
                    // A badly formed substring will throw an
                    // ArrayIndexOutOfBoundsException
                    for (lastPos = firstPos + 1;
                                lastPos < nameChar.length; lastPos++) {
                        if (nameChar[lastPos] == endChar)
                            break;
                    }
                    // If the lastPos is a quote, we need to
                    // include it in the string; if it's a comma
                    // we don't
                    return new String(nameChar, firstPos,
                            (endChar == ',' ?
                                lastPos - firstPos :
                                lastPos - firstPos + 1));
                }
                // else try the next index
            }
        }
        return null;
    }

    public String toString(  ) {
        getCN(  );
        getOU(  );
        getO(  );
        getL(  );
        getST(  );
        getC(  );
        return "CN=" + CN + ", " +
               "OU=" + OU + ", " +
               "O=" + O + ", " +
               "L=" + L + ", " +
               "ST=" + ST + ", " +
               "C=" + C;
    }
}
```

Given an X500 name, this class can extract any of the six standard fields from it when required. Because not all X500 names have all the possible fields, we parse the entire string for a field that we're interested in.

14.1.3 JSSE Socket Factories

The JSSE API defines a set of factory classes that are used to obtain sockets. These factories are independent of SSL; in fact, they reside in the `javax.net` package rather than the `javax.net.ssl` package (although both packages are part of JSSE). You can use these factories to obtain sockets that use SSL or to obtain plain TCP sockets (instead of directly using the constructors of the `Socket` and `ServerSocket` classes).

This adds a nice level of abstraction to network code: the code can switch from plain to SSL sockets simply by changing socket factories. There are often times when you need to perform special initialization operations on SSL sockets, in which case this abstraction doesn't really help. But for simple cases, the factory classes are a nice feature.

The `javax.net` package defines two types of factories: the `SocketFactory` class (`javax.net.SocketFactory`) and the `ServerSocketFactory` class (`javax.net.ServerSocketFactory`). The default implementations of these classes produce plain sockets; their subclasses will produce SSL sockets (or, potentially, sockets that use other protocols).

The API of the `ServerSocketFactory` class provides the following two methods:

public ServerSocket createServerSocket(int port)
public ServerSocket createServerSocket(int port, int backlog)
public ServerSocket createServerSocket(int port, int backlog, InetAddress addr)

Create a server socket that listens on the given port. If no backlog is specified, a system default value (50) is used. On machines with multiple network interfaces, the `addr` parameter may be used to specify that the socket should listen only on the given interface. If the socket cannot be created, an `IOException` is thrown.

public static ServerSocketFactory getDefault( )

Return the default plain server socket factory for the platform. Subclasses of the `ServerSocketFactory` class override this method to provide a factory that produces sockets that use the desired protocol.

Similarly, the `SocketFactory` class provides the following methods:

public Socket createSocket(String host, int port)
public Socket createSocket(InetAddress host, int port)
public Socket createSocket(String host, int port, InetAddress clientHost, int clientPort)
public Socket createSocket(InetAddress address, int port, InetAddress client, int clientPort)

Create a socket that connects to the given host on the given port. If the client has multiple network interfaces, the last two methods may be used to select which interface to use on the client; they may also be used to specify which port should be used on the client. If no port is specified, an anonymous port is used.

If the given host is unknown, an `UnknownHostException` is thrown. If the socket cannot be created, an `IOException` is thrown.

public static ServerSocketFactory getDefault( )

Return the default plain socket factory for the platform. Subclasses of the `SocketFactory` class override this method to provide a factory that produces sockets that use the desired protocol.

We'll show how these factories are used in the next few sections.

14.2 SSL Client and Server Sockets

In this section, we'll explore how SSL sockets are created and develop a simple server and client that can be used to exchange data over an SSL connection. As we proceed further in this chapter, we'll modify these simple programs to take advantage of more advanced SSL features.

14.2.1 SSL Server Sockets

SSL server sockets are obtained through the `SSLServerSocketFactory` class (`javax.net.ssl.SSLServerSocketFactory`), which extends the `Server-SocketFactory` class. The `SSLServerSocketFactory` class overrides the `getDefault( )` method to provide a class that produces SSL server sockets:

public static ServerSocketFactory getDefault( )

Return the default SSL server socket factory implementation. That factory can be used to obtain SSL server sockets. The default implementation is defined in the *$JREHOME/lib/security/java.security* file by the property `ssl.ServerSocketFactory.provider`. If this is not set (by default, it is not), a hardwired, internal implementation is used (the class `com.sun.net.ssl.internal.ssl.SSLServerSocketFactoryImpl`).

Note that the `ssl.ServerSocketFactory.provider` property is ignored in the exportable version of JSSE; you can use a different implementation of the socket factory only in the version of JSSE available in the U.S. and Canada. Even though the export restrictions for JSSE have been relaxed, they still do not permit users in most of the world to substitute their own SSL implementations.

The default socket factory will handle both SSL 3.0 and TLS 1.0 protocols.

Here's our first implementation of a server:

```
package javasec.samples.ch14;

import java.io.*;
import java.net.*;
import java.security.*;

import javax.net.*;
import javax.net.ssl.*;

public class SSLSimpleServer extends Thread {
    public static void main(String[] args) throws Exception {
        ServerSocketFactory ssf = SSLServerSocketFactory.getDefault(  );
        ServerSocket ss = ssf.createServerSocket(9096);

        while (true) {
            new SSLSimpleServer(ss.accept()).start(  );
        }
    }

    private Socket sock;

    public SSLSimpleServer(Socket s) {
        sock = s;
    }

    public void run(  ) {
        try {
            BufferedReader br = new BufferedReader(
                                    new InputStreamReader(
                                        sock.getInputStream(  )));
            PrintWriter pw = new PrintWriter(sock.getOutputStream(  ));

            String data = br.readLine(  );
            pw.println("What is she?");
            pw.close(  );
            sock.close(  );
        } catch (IOException ioe) {
            // Client disconnected; exit this thread
        }
    }
}
```

Other than the new way in which we've constructed the server socket, this is standard Java socket code: each connection is handled in a separate thread. The server expects to read one line of data and write a one−line reply. Note that we used the `SSLServerSocketFactory` class to provide the default factory; if we wanted to run this code with plain sockets, we'd change the first line of the `main( )` method to use the `ServerSocketFactory` class instead.

The socket returned from the `SSLServerSocketFactory` will be an instance of the `SSLServerSocket` class (`javax.net.ssl.SSLServerSocket`), which extends the `ServerSocket` class. Some additional limited operations can be performed on the SSL server socket, but you generally treat it as you would any other server socket.

To run this example, use the keystore we created earlier and run this command:

```
piccolo% java −Djavax.net.ssl.keyStore=$HOME/.keystore \
              −Djavax.net.ssl.keyStorePassword=****** \
              javasec.samples.ch14.SSLSimpleServer
```

This server will run indefinitely, sending out a line of text to each client that connects to it. Although it is not a real HTTPS server, you can connect to it from most browsers using the URL *https://localhost:9096/* (though the browser will tell you it doesn't recognize the certificate). You can also test it with the sample SSL clients we develop throughout the rest of this chapter.

14.2.2 SSL Sockets

SSL client sockets are obtained from the `SSLSocketFactory` class (`javax.net.ssl.SSLSocketFactory`), which extends the `SocketFactory` class (`javax.net.SocketFactory`). The `SSLSocketFactory` class overrides the `getDefault( )` method to provide a factory that produces SSL sockets:

public static SocketFactory getDefault( )

Obtain the default SSL socket factory for this implementation. That factory can be used to obtain SSL sockets. The default implementation is defined in the *$JREHOME/lib/security/java.security* file by the property `ssl.SocketFactory.provider`. If this is not set (by default, it is not), a hardwired, internal implementation is used.

Like its server analogue, the `ssl.SocketFactory.provider` property is ignored in the exportable version of JSSE. The default socket factory will handle both SSL 3.0 and TLS 1.0 protocols.

Here's how we can write a simple client:

```
package javasec.samples.ch14;

import java.io.*;
import java.net.*;

import javax.net.*;
import javax.net.ssl.*;

public class SSLSimpleClient {

    public static void main(String[] args) throws Exception {
        SocketFactory sf = SSLSocketFactory.getDefault(  );
        Socket s = sf.createSocket(args[0], Integer.parseInt(args[1]));

        BufferedReader br = new BufferedReader(
                                    new InputStreamReader(
                                        s.getInputStream(  )));
        PrintWriter pw = new PrintWriter(s.getOutputStream(  ));
        System.out.println("Who is Sylvia?");
        pw.println("Who is Sylvia?");
        pw.flush(  );
        System.out.println(br.readLine(  ));
        s.close(  );
    }
}
```

Note again that we used the `SSLSocketFactory` class to obtain the socket factory; to use this client to connect to a non−SSL server, we'd change the first line of the `main( )` method to use the `SocketFactory` class instead.

The socket returned from the SSL socket factory will be an instance of the `SSLSocket` class (`javax.net.ssl.SSLSocket`), which extends the `Socket` class. For the most part you can treat it like any other socket, but we will look at some advanced ways of handling the SSL socket a little later.

To run this program, we must supply the host and port that we want to contact. More importantly, the server will present its certificate to us, and we must have the root certificate of the server's CA in our truststore.

Using the truststore we created earlier, we can run the client with this command:

```
piccolo% java −Djavax.net.ssl.trustStore=$HOME/.truststore \
              javasec.samples.ch14.SSLSimpleClient localhost 9096
Who is Sylvia?
What is she?
```

The first line of output is the string we sent to the server; the second line is the string returned by the server.

At this point, we've developed a very simple server and client that communicate via SSL. We've relied on the SSL implementation to provide many of the details of this communication for us: the certificates used for verification, the cipher suites used, and so on. In the next few sections, we'll look at the SSL support classes that allow us to modify these things.

14.3 SSL Sessions

When an SSL socket is connected, it joins an SSL session. Every SSL socket belongs to one SSL session. If there are many connections between a client and server they may share an SSL session, but there is no programmatic way to determine which sockets are attached to which sessions. From our perspective, the SSL session is just an object that allows us to retrieve certain information about a particular SSL connection. Specifically, the SSL session allows us to perform a necessary (but often ignored) step in the verification of an SSL peer.

In our first example, the server has sent its certificate to the client. If the client recognized the CA that issued the server's certificate, we allowed communication to proceed. However, as we've mentioned before, just because someone has a valid certificate does not necessarily mean that they can be trusted. I can get a valid certificate for *www.myevilenterprise.com* and then spoof your DNS server so that when you attempt to connect to *shopping.oreilly.com*, you get connected to me. Just because I present a certificate to you is no assurance that I am authorized to take your credit card information.

So when you make an SSL connection, you should verify that the server certificate contains the information you expect. Typically, this means that the name within the certificate should be the domain name of the machine to which you're connecting. If your server requires clients to authenticate themselves, the server should follow this procedure with the client's certificate as well.

To verify this, you must use an object that implements the `SSLSession` interface (`javax.net.ssl.SSLSession`). SSL session objects are retrieved from the `SSLSocket` class using this method:

public SSLSession getSession( )
: Retrieve the SSL session associated with this SSL socket.

Because this method operates on an SSL socket (and not a generic socket), we'll have to modify the protocol−independent technique we used in our first client in order to use it. Now when we get a socket back from the `SSLSocketFactory`, we'll have to cast it to an `SSLSocket` object. Nonetheless, verifying the name in the server's certificate is important enough to warrant the protocol−specific code.

The `SSLSession` interface contains the following methods:

public byte[] getId( )
: Return the arbitrary identifier assigned to this session.

public SSLSessionContext getSessionContext( )
: Return the session context associated with this session. The session context simply groups together multiple SSL sessions; we won't use it in our examples.

public long getCreationTime( )
: Return the time at which this SSL session was created.

public long getLastAccessedTime( )
: Return the time at which this SSL session was last accessed (i.e., when a new connection was last added to the session).

public void invalidate( )
: Invalidate this session. Existing connections will not be affected by this call; new connections will need to join a new session.

public void putValue(String name, Object value)
: Bind an object into this session, associated with the given name.

public Object getValue(String name)
: Return the bound object associated with the given name.

public void removeValue(String name)
: Remove the bound object associated with the given name.

public String[] getValueNames( )
: Retrieve the names of all bound objects.

public javax.security.cert.X509Certificate[] getPeerCertificateChain( )
: Retrieve the certificate chain that the peer sent in order to verify itself to us. This method throws an `SSLPeerUnverifiedException` in two cases: if the server executes this method but does not require the client to authenticate itself, and if the certificates presented by the peer are not recognized (that is, if the root certificate does not correspond to a certificate held in the truststore).

public String getCipherSuite( )
: Return the name of the cipher suite that the peers negotiated.

public String getPeerHost( )
: Return the name of the host to which this socket is connected.

Unless you're writing your own implementation of JSSE, the last three methods are really the only ones that are useful. The SSL protocol defines a technique by which SSL connections can be reestablished without a full (time−consuming) handshake, and SSL sessions can be used to provide that. However, that detail is dependent on the JSSE implementation and is not something we can change programatically.

We will use the last three methods to modify our original client. We can use the relevant methods of the SSL session to retrieve information and complete the verification of the server. Changes to our original client are shown in bold:

```
package javasec.samples.ch14;

import java.io.*;
import java.net.*;

import javax.net.*;
import javax.net.ssl.*;
import javax.security.cert.*;

public class SSLClientVerifier {

    public static void main(String[] args) throws Exception {

        SocketFactory sf = SSLSocketFactory.getDefault(  );
        SSLSocket s = (SSLSocket) sf.createSocket(
                       args[0], Integer.parseInt(args[1]));

        SSLSession sess = s.getSession(  );
        String host = sess.getPeerHost(  );
        X509Certificate[] certs = sess.getPeerCertificateChain(  );
        String dn = certs[0].getSubjectDN().getName(  );
        X500Name name = new X500Name(dn);
        if (!host.equals(name.getCN(  )))
            System.err.println("Warning: Expected " + host +
                               " and got " + name.getCN(  ));

        BufferedReader br = new BufferedReader(
                                   new InputStreamReader(
                                       s.getInputStream(  )));
        PrintWriter pw = new PrintWriter(s.getOutputStream(  ));
        System.out.println("Who is Sylvia?");
        pw.println("Who is Sylvia?");
        pw.flush(  );
        System.out.println(br.readLine(  ));
        s.close(  );
    }
}
```

We use the session object to retrieve the server's certificate. If the common name in the server's certificate is not equal to its hostname, we print out a warning message. It is conceivable that the name we used to construct the socket may not exactly match the name embedded within the certificate, so we should obtain the IP addresses of both and ensure that they are the same. That's the technique we'll use a little later when we perform HTTP client verification. Note that this is the same thing a browser does when the name within a site certificate does not match the name within the URL.

If you run this with the test keystore created earlier you'll see the warning message, but if you create another keystore with the common name equal to the hostname of your server, this code will execute without the warning.

14.4 SSL Contexts and Key Managers

The generic SSL sockets utilize several default options, including the protocol version used (TLS 1.0, which also supports SSL 3.0) and the default key and trust managers. If you want to change any of these options, you must use an instance of the `SSLContext` class (`com.sun.net.ssl.SSLContext`). This is often necessary, although it moves us away from a strictly protocol−independent socket factory paradigm.

The `SSLContext` class is an engine class; it provides the following methods:

public static SSLContext getInstance(String protocol)
public static SSLContext getInstance(String protocol, Provider provider)
public static SSLContext getInstance(String protocol, String provider)

Obtain an SSL context that implements the given protocol, optionally by consulting only the given provider. In Sun's implementation of JSSE, the protocol must be TLS, TLSv1, SSL, or SSLv3. Since later versions of the protocol can support earlier versions, the string TLS provides the most generic instance of an SSL context.

If the given algorithm cannot be found, a `NoSuchAlgorithmException` is thrown. If the given provider cannot be found, a `NoSuchProviderException` is thrown.

public final String getProtocol( )

Return the protocol supported by this context.

public final Provider getProvider( )

Return the provider that supplied the implementation of this context.

public final void init(KeyManager[] km, TrustManager[] tm, SecureRandom random)

Initialize this context with the given key managers, trust managers, and random number generator. Any or all of these parameters may be `null`, in which case default, internal implementations are used. If the context cannot be initialized, a `KeyManagementException` is thrown.

public final SSLSocketFactory getSocketFactory( )

Return a factory that produces SSL sockets based on the parameters of this context.

public final SSLServerSocketFactory getServerSocketFactory( )

Return a factory that produces SSL server sockets based on the parameters of this context.

Typically, you use an SSL context so you can specify a new key manager or trust manager in the `init( )` method. We'll see how to do that next.

14.4.1 Working with Key Managers

In our first server, we used the default key manager. The default key manager reads the keystore specified by the `javax.net.ssl.keyStore` property and finds the first alias in that keystore that has a key appropriate for use with the algorithm the server socket wants to use. Details on how the server socket selects an algorithm are given later in this chapter, but in essence the server will find the first RSA key or the first DSA key, and so on. The alias that is chosen is arbitrary, as it depends upon the order in which the aliases are returned via a hashtable enumeration.

If you want to use a different keystore, or if you want to make sure that you use a particular alias, you must interact with the key manager. In the first case, you can simply initialize the default key manager with an appropriate keystore. In the second case, you must write your own key manager. We'll look at both of those cases now.

The default key manager is obtained via the `KeyManagerFactory` class (`com.sun.net.ssl.KeyManagerFactory`), which contains the following methods:

public static final String getDefaultAlgorithm( )
: Return the default algorithm for key managers. In Sun's implementation of JSSE, this returns the string "SunX509". You can change the default algorithm by changing the `sun.ssl.keymanager.type` property in the *$JREHOME/lib/security/java.security* file (in which case you must have a third−party security provider that implements that algorithm).

public final String getAlgorithm( )
: Return the name of the algorithm that this instance of the key manager factory implements.

public static final KeyManagerFactory getInstance(String algorithm)
public static final KeyManagerFactory getInstance(String algorithm, Provider provider)
public static final KeyManagerFactory getInstance(String algorithm, String provider)
: Obtain a key manager factory that implements the given algorithm, optionally by consulting only the given provider. With the security provider that comes with Sun's implementation of JSSE, the algorithm must be "SunX509". A `NoSuchAlgorithmException` is thrown if the algorithm cannot be found; if the provider cannot be found a `NoSuchProviderException` is thrown.

public final Provider getProvider( )
: Return the provider that supplied this key manager factory.

public void init(KeyStore ks, char[] password)
: Initialize the key manager factory so that it obtains keys from the given keystore using the given password. Note that only one password is used in this method, so the keystore must use the same password for all the private keys it stores. Initialization may throw a `KeyStoreException`, a `NoSuchAlgorithm−Exception`, or an `UnrecoverableKeyException`.

public KeyManager[] getKeyManagers( )
: Obtain the key managers that this factory supplies.

These last two methods are the most important ones for our consideration. The `init( )` method allows us to specify a different keystore, and the `getKey−Managers( )` method gets the actual key managers that the SSL classes need.

Here's how we can extend our original server so that it uses a different keystore than the one specified via the command−line property (changes are again shown in bold):

```
package javasec.samples.ch14;

import java.io.*;
import java.net.*;
import java.security.*;

import javax.net.*;
import javax.net.ssl.*;
import com.sun.net.ssl.*;
```

```
public class SSLServerWithContext {

    public static void main(String[] args) throws Exception {
        SSLContext sc = SSLContext.getInstance("TLS");
        KeyManagerFactory kmf =
                        KeyManagerFactory.getInstance("SunX509");
        KeyStore ks = KeyStore.getInstance("jceks");
        char[] password = args[1].toCharArray(  );
        ks.load(new FileInputStream(args[0]), null);
        kmf.init(ks, password);
        sc.init(kmf.getKeyManagers(  ), null, null);

        ServerSocketFactory ssf = sc.getServerSocketFactory(  );
        ServerSocket ss = ssf.createServerSocket(9096);

        while (true) {
            new SSLSimpleServer(ss.accept()).start(  );
        }
    }
}
```

Here, we create the SSL context and then obtain a key manager factory. The factory is initialized with the keystore that we load from the first argument on the command line. The keystore uses the second argument to look up its private keys. Once the context is initialized, we create the server socket factory from the context and obtain the server socket from that factory. As before, the server socket is actually an instance of the `SSLServerSocket` class, and the bulk of the program logic is still provided by the `SSLSimpleServer` class.

In this case, all we've really done is gone from specifying the keystore via a property to specifying it on the command line. This server is run as follows:

```
piccolo% java javasec.samples.ch14.SSLServerWithContext $HOME/.keystore ******
```

This code contains the outline to use if you have a nondefault keystore; notice, for instance, that we're using the JCEKS algorithm as input to the `getInstance( )` method of the `KeyStore` class. Similarly, if there are other differences in the way you obtain keystores, or if you'd rather not specify the command−line property, this code shows how to initialize the key manager.

However, our server still will use an arbitrary alias from the keystore. So now we'll take this example one step further and develop a key manager that allows us to specify which alias in the keystore to use.

To write our own key manager, we must develop a class that implements the `X509KeyManager` interface (`com.sun.net.ssl.X509KeyManager`). This interface extends the `KeyManager` interface (`com.sun.net.ssl.KeyManager`), which is an empty interface used for type identification. Because the `KeyManager` interface does not define an API, the implementation of a key manager is tightly coupled with the implementation of the `SSLContext` class in use. Sun's implementation of the `SSLContext` class requires you to pass at least one key manager that implements the `X509KeyManager` interface in the key manager array passed to the `init( )` method; the first such class in the array is the one that the `SSLContext` class uses to look up keys. All other key managers in the array are ignored. If no appropriate key manager is present in the array no exception is thrown, but nothing will work either. If you're using a third−party security provider that defines a different `SSLContext` class, it may expect a different type of key manager.

To write our key manager, we must implement the following methods of the `X509KeyManager` interface:

public String[] getClientAliases(String keyType, Principal[] issuers)

Return all possible aliases (from the keystore) that could be used to perform client−side SSL encryption for the given key type (e.g., RSA or DSA). If the `issuers` array is not `null`, it will contain an array of principals who are CAs. The key belonging to an alias in the returned array must have been issued by an entity contained in the array.

This method should not return `null`; if no appropriate aliases are found, it should return an array of length 0.

public String chooseClientAlias(String keyType, Principal[] issuers)
Select the alias (from the keystore) that should be used to perform client−side SSL encryption for the given key type (e.g., RSA or DSA). If the `issuers` array is not `null`, the key must be provided by an entity contained in the array. This method should return `null` if no appropriate alias is found.

public String[] getServerAliases(String keyType, Principal[] issuers)
Return all possible aliases (from the keystore) that could be used to perform server−side SSL encryption for the given key type (e.g., RSA or DSA). If the `issuers` array is not `null`, the key must be provided by an entity contained in the array.

This method should not return `null`; if no appropriate aliases are found, it should return an array of length 0.

public String chooseServerAlias(String keyType, Principal[] issuers)
Select the alias (from the keystore) that should be used to perform server−side SSL encryption for the given key type (e.g., RSA or DSA). If the `issuers` array is not `null`, the key must be provided by an entity contained in the array. This method should return `null` if no appropriate alias is found.

public X509Certificate[] getCertificateChain(String alias)
Return the array of X509 certificates associated with the given alias. If the keystore contains non−X509 certificates for the given alias, those certificates should be ignored. Note that the method returns an array of `java.security.cert.X509Certificate` objects, not (as do most other classes within JSSE) `javax.security.cert.X509Certificate` objects.

public PrivateKey getPrivateKey(String alias)
Return the private key for the alias.

As the API indicates, key managers may be used by both clients and servers in an SSL conversation. They are more frequently used by servers, since servers must always authenticate themselves to clients.

Here's an implementation of this interface:

```
package javasec.samples.ch14;

import java.net.*;
import java.security.cert.X509Certificate;
import java.security.interfaces.*;
import java.security.*;
import java.util.*;

import javax.net.ssl.*;
import com.sun.net.ssl.*;

public class SSLKeyManager implements X509KeyManager {
    protected String alias;
    protected KeyStore ks;
    protected char[] pw;
    private String type;
    private String issuer;

    SSLKeyManager(KeyStore ks, String s, char[] pw) {
        this.ks = ks;
        alias = s;
        this.pw = pw;
        try {
            java.security.cert.Certificate c = ks.getCertificate(s);
            type = c.getPublicKey().getAlgorithm(  );
            issuer = ((X509Certificate) c).getIssuerDN().getName(  );
        } catch (Exception e) {
            throw new IllegalArgumentException(s + " has a bad key");
        }
    }

    public String chooseClientAlias(String type, Principal[] issuers) {
        if (!type.equals(this.type))
            return null;
        if (issuers == null)
            return alias;
        for (int i = 0; i < issuers.length; i++) {
            if (issuer.equals(issuers[i].getName(  )))
                return alias;
        }
        return null;
    }

    public String chooseServerAlias(String type, Principal[] issuers) {
        return chooseClientAlias(type, issuers);
    }

    // Get the certificates -- make sure each is an X509Certificate
    // before copying it into the array.
    public X509Certificate[] getCertificateChain(String s) {
        try {
            java.security.cert.Certificate[] c =
                                    ks.getCertificateChain(s);
            Vector c2 = new Vector(c.length);
            for (int i = 0; i < c.length; i++)
                c2.add(c[i]);
            return (X509Certificate[])
                    c2.toArray(new X509Certificate[0]);
        } catch (KeyStoreException kse) {
            return null;
        }
    }

    public String[] getClientAliases(String type, Principal[] p) {
        String[] s;
        String alias = chooseClientAlias(type, p);
        if (alias == null)
            s = new String[0];
        else {
            s = new String[1];
            s[0] = alias;
        }
        return s;
    }

    public String[] getServerAliases(String type, Principal[] p) {
        return getClientAliases(type, p);
    }
    public PrivateKey getPrivateKey(String alias) {
        try {
            return (PrivateKey) ks.getKey(alias, pw);
        } catch (Exception e) {
            return null;
        }
    }
}
```

This key manager is initialized with a keystore and the particular alias in the keystore that you want to use. Since we've specified an alias we treat client and server authorization the same way, though you could extend this idea to provide different aliases or otherwise treat the server authentication differently.

As a result of this simplification, most methods end up calling the `choose-ClientAlias( )` method. This method checks to see if the key algorithm type matches and, if appropriate, if the key was provided by an entry in the `issuers` array. If everything matches, it returns the alias we want; otherwise it returns `null`.

The key manager itself must come from a key manager factory.[1] Hence, the next step we must take is to write a class that extends the `KeyManagerFactorySpi` class (`com.sun.net.ssl.KeyManagerFactorySpi`). That class contains the following abstract methods:

[1] Strictly speaking, this isn't true: we could instantiate the `SSLKeyManager` object directly, create an array for it, and pass that to the `init( )` method of the SSL context. But that violates the spirit of the Java security framework.

protected abstract void engineInit(KeyStore ks, char[] password)

Initialize the key manager factory, using the given keystore and password. This method may throw a `KeyStoreException`, a `NoSuchAlgorithmException`, or an `UnrecoverableKeyException`, as necessary.

protected abstract KeyManager[] engineGetKeyManagers( )

Return the array of key managers that work with the initialized keystore and password.

Here's how we implement the engine:

```
package javasec.samples.ch14;

import java.security.*;
import javax.net.ssl.*;
import com.sun.net.ssl.*;

public class SSLKeyManagerFactory extends KeyManagerFactorySpi {
    char[] pw;
    KeyStore ks;
    String alias;

    public SSLKeyManagerFactory(  ) {
        alias = System.getProperty("xyz.aliasName");
        if (alias == null)
            throw new IllegalArgumentException(
                                 "Must specify alias property");
    }

    protected KeyManager[] engineGetKeyManagers(  ) {
        SSLKeyManager[] km = new SSLKeyManager[1];
        km[0] = new SSLKeyManager(ks, alias, pw);
        return km;
    }

    protected void engineInit(KeyStore ks, char[] pw) {
        this.ks = ks;
        this.pw = new char[pw.length];
        System.arraycopy(pw, 0, this.pw, 0, pw.length);
    }
}
```

The key manager factory expects a keystore and a password but not an alias, so we've had to specify the alias via a system property. In general, if you need other initialization fields in your own key manager, this is the way to get them.

This engine class requires a security provider, of course; we'll use the provider from Chapter 8, which contains this mapping:

```
put("KeyManagerFactory.XYZ",
                  "javasec.samples.ch14.SSLKeyManagerFactory");
```

Finally, here's how we use the key manager within our server:

```
package javasec.samples.ch14;

import java.io.*;
import java.net.*;
import java.security.*;

import javax.net.*;
import javax.net.ssl.*;
import javax.security.cert.*;
import com.sun.net.ssl.*;
import javasec.sample.ch08.XYZProvider;

public class SSLServerKeyManager {
    public static void main(String[] args) throws Exception {
        Security.addProvider(new XYZProvider(  ));

        SSLContext sc = SSLContext.getInstance("TLS");
        KeyStore ks = KeyStore.getInstance("jceks");
        char[] password = args[1].toCharArray(  );
        ks.load(new FileInputStream(args[0]), null);

        KeyManagerFactory kmf = KeyManagerFactory.getInstance("XYZ");
        kmf.init(ks, password);
        sc.init(kmf.getKeyManagers(  ), null, null);

        ServerSocketFactory ssf = sc.getServerSocketFactory(  );
        ServerSocket ss = ssf.createServerSocket(9096);

        while (true) {
            new SSLSimpleServer(ss.accept()).start(  );
        }
    }
}
```

The only code change here is that we've installed the security provider and used the XYZ algorithm to provide the key manager factory. However, when we run this version of the server, we must remember to set the `xyz.aliasName` property to an entry in the keystore that we're loading:

```
piccolo% java −Dxyz.aliasName=sdo javasec.samples.ch14.SSLServerKeyManager \
                    $HOME/.keystore ******
```

14.4.2 Working with Trust Managers

There is less need to provide a custom trust manager than to provide a custom key manager. Unlike the key manager, the trust manager is never asked to provide a specific alias or single certificate: it is asked to provide all the certificates it can use to validate a peer, and it is asked to validate peers. Hence, you might provide a custom trust manager if you want to limit the certificates from a truststore that you use to validate peers or if you want to do some sort of special validation of a peer (such as checking an external certificate−revocation list).

Trust managers generally are provided from trust manager factories. The `TrustManagerFactory` class (`com.sun.net.ssl.TrustManagerFactory`) is completely analogous to the `KeyManagerFactory` class: it contains a static `getDefaultAlgorithm( )` method that returns the `sun.ssl.trustmanager.type` property defined in the *$JREHOME/lib/security/java.security* file. By default that property is not defined, and the algorithm defaults to SunX509.

The static `getInstance( )` methods of the `TrustManagerFactory` class provide specific trust manager factories, and the `getTrustManager( )` method returns an array of trust managers that a specific factory defines.

Similarity to the `KeyManagerFactory` interface extends to the definition of the trust manager itself. The `TrustManager` interface (`com.sun.net.ssl.TrustManager`) is an empty interface, and Sun's implementation of the `SSLContext` class requires that the trust manager actually implement the `X509TrustManager` interface (`com.sun.net.ssl.X509TrustManager`).

We'll briefly discuss the `X509TrustManager` interface here. It's a simple enough interface that we won't show a complete example that uses it. The interface contains the following three methods:

public boolean isClientTrusted(X509Certificate[] chain)
: Check that the chain of certificates provided by the peer can be validated by root certificates in the truststore and that the validation is trusted for client−side SSL (that is, your server should trust the connecting client).

public boolean isServerTrusted(X509Certificate[] chain)
: Check that the chain of certificates provided by the peer can be validated by root certificates in the truststore and that the validation is trusted for server−side SSL (that is, whether your client should trust the server to which you are connecting).

public X509Certificate[] getAcceptedIssuers( )
: Return the list of certificates you will use to validate the peer. This is normally all the root certificates in the truststore.

14.5 Miscellaneous SSL Issues

Finally, there are a number of miscellaneous SSL issues that the SSL socket API is designed to handle, including SSL proxies, client−side authentication, choosing a cipher suite, SSL handshaking, and JSSE permissions.

14.5.1 SSL Proxies

SSL clients often need to make connections through a proxy server; this enables them to make requests through a firewall. If you need to make a connection through a proxy server, use this method of the `SSLSocketFactory` class:

public abstract Socket createSocket(Socket s, String host, int port, boolean autoClose)

Create an SSL socket to the given host and port that uses the existing socket as its proxy. The existing socket is a standard (plain) socket that has been connected to the appropriate proxy host and proxy port. If `autoClose` is `true`, the underlying socket will be closed when this socket is closed. If the socket cannot be created, an `IOException` is thrown.

If you're using your own protocol, it's up to you to define what data should flow between your program and the proxy server before layering the sockets with this call. If you're using HTTPS, you must send a connect string and read the headers from the proxy server on the underlying socket before you create the SSL socket. JSSE comes with a set of sample code that shows how this can be accomplished. However, if you're using HTTPS as your protocol, it's far easier to use the HTTPS protocol handler, which handles all these details for you (see Section 14.6 later in this chapter).

14.5.2 Client−Side Authentication

As we've mentioned, in most SSL conversations the server presents credentials to the client and the client verifies those credentials; the client is then assured of the server's identity. The client does not normally present its certificate credentials to the server; in fact, the client is not even required to possess such credentials.

SSL servers can require the client authenticate itself as well, however. To do this, the server uses the `setNeedClientAuth( )` method on its server socket as follows:

```
SSLContext sc = ... set up context as before ...
SSLServerSocketFactory sssf =
      (SSLServerSocketFactory) sc.getServerSocketFactory(  );
SSLServerSocket sss = (SSLServerSocket) sssf.createServerSocket(9096);
sss.setNeedClientAuth(true);
```

Note that this leads us further from the path of protocol−independent socket factories. Although the `createServerSocket( )` method returns an instance of the `ServerSocket` class, we must cast that object to an `SSLServerSocket` in order to call the `setNeedClientAuth( )` method.

Remember that if you put this code into your server, your server will need a valid truststore that contains the root certificate of the client's certificate chain. Be sure to set the appropriate `trustStore` property when you run a server with this code.

14.5.3 Choosing an SSL Cipher Suite

When an SSL conversation begins, the client and server negotiate between themselves as to which SSL cipher suite they will use. The suite is chosen based upon the credentials that each side possesses and the suites that each side supports. For example, a server can't support an RSA cipher suite unless it has an available RSA private key. The client and server must support at least one common cipher suite in order to communicate; if they both support multiple ciphers, the strongest available suite will be chosen.

SSL sockets in Sun's implementation of JSSE support 15 different cipher suites, as listed in Table 14−1. These strings are part of the SSL specification and are defined as `SSL_<key exchange`

`algorithm>_WITH_<encryption algorithm>_<hash algorithm>`. When a number appears in the encryption algorithm, it refers to the key strength of the encryption: higher numbers are more secure (though the highest numbers used to be subject to export control, and many other SSL implementations may not support them).

Table 14−1. SSL Cipher Suites Supported by Sun's JSSE Implementation

Prefix	Key Exchange	Encryption	Hash
SSL_	DH_anon_EXPORT_	WITH_DES40_CBC_	SHA
SSL_	DH_anon_EXPORT_	WITH_RC4_40_	MD5
SSL_	DH_anon_	WITH_3DES_EDE_CBC_	SHA
SSL_	DH_anon_	WITH_DES_CBC_	SHA
SSL_	DH_anon_		

WITH_RC4_128_

MD5

SSL_

DHE_DSS_EXPORT_

WITH_DES40_CBC_

SHA

SSL_

DHE_DSS_

WITH_3DES_EDE_CBC_

SHA

SSL_

DHE_DSS_

WITH_DES_CBC_

SHA

SSL_

RSA_EXPORT_

WITH_RC4_40_

MD5

SSL_

RSA_

WITH_3DES_EDE_CBC_

SHA

SSL_

RSA_

WITH_DES_CBC_

SHA

SSL_

RSA_

WITH_NULL_

MD5

SSL_

RSA_

WITH_NULL_

SHA

SSL_

RSA_

WITH_RC4_128_

MD5

SSL_

RSA_

WITH_RC4_128_

SHA

Even though these 15 suites are all supported, by default just 8 of them are enabled; those suites are listed in bold in the table. The other suites will not be used unless you enable them explicitly.

To see and modify the suites that an SSL socket is using, use the following methods (which are defined for both the `SSLSocket` and `SSLServerSocket` classes):

public abstract String[] getEnabledCipherSuites( )
: Return an array containing all the cipher suites that are enabled on the socket. Unless different suites have been enabled, this will return an array containing the eight bold items in Table 14−1.

public abstract void setEnabledCipherSuites(String[] suites)
: Set which cipher suites are enabled on the particular socket. The list completely overwrites the old set of enabled suites; if you want to add support for a new suite, you must construct an array containing all the presently enabled suites plus the new suite you want to support. Each string in the given array must have been listed in the array returned by the `getSupportedCipherSuites( )` method.

public abstract String[] getSupportedCipherSuites( )
: Return an array containing all the cipher suites that this implementation supports.

To add support for anonymous, exportable Diffie−Hellman key exchange using 40−bit DES encryption and an

SHA hash to a server socket, execute this code:

```
SSLServerSocket sss = ... initialize from socket factory ...;
String[] enabled = sss.getEnabledCipherSuites(  );
String[] newEnabled = new String[enabled.length + 1];
System.arraycopy(enabled, 0, newEnabled, 0, enabled.length);
newEnabled[enabled.length] = "SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA";
sss.setEnabledCipherSuites(newEnabled);
```

If you're writing the client, you must supply similar code for the `SSLSocket`.

14.5.4 SSL Handshaking

When the SSL client and server negotiate which cipher suite to use, they engage in an SSL handshake. If you want to keep track of the handshake, you can use various handshaking event classes.

A handshake event is like any other event. To receive a handshake event, register a class that implements the `HandshakeCompletedListener` interface (`javax.net.ssl.HandshakeCompletedListener`) by calling the `addHandshake−CompletedListener( )` method of the SSL socket. For SSL servers, call this method on the socket returned from the `accept( )` method of the SSL server socket; it is defined only in the `SSLSocket` class. The listener can be removed with the `removeHandshakeCompletedListener( )` method.

When the handshake is complete, the `handshakeCompleted( )` method of the handshake listener is called and it is passed a `HandshakeCompletedEvent` object (`javax.net.ssl.HandshakeCompletedEvent`). This event allows you to retrieve the cipher suite that was negotiated, the certificate chain the peer presented for authentication (if applicable), the SSL session that is in use, and the socket on which the event occurred. Note that all of this information is available directly from the SSL socket as well.

14.5.5 JSSE Permissions

To use some features of JSSE in an environment in which a security manager has been installed, untrusted code will need the following permissions:

```
permission com.sun.net.ssl.SSLPermission "setHostnameVerifier";
permission com.sun.net.ssl.SSLPermission "setDefaultAuthenticator";
permission com.sun.net.ssl.SSLPermission "getSSLSessionContext";
```

The `setHostnameVerifier` permission is needed to call the `setDefault−HostnameVerifier( )` method of the `HttpsURLConnection` class. It is not needed to call the `setHostnameVerifier( )` method of that class; setting the global verifier is the only option considered unsafe. The `setDefaultAuthenticator` permission is needed to call the `setDefaultAuthenticator( )` method of the `HttpsURLConnection` class. The `getSSLSessionContext` permission is needed to call the `getSessionContext( )` method of the `SSLSession` class.

14.6 The HTTPS Protocol Handler

SSL is often used as the underlying communication protocol of HTTPS. If you're talking to an HTTPS server you can write the SSL−level code yourself, but it's generally easier to use the standard URL class to talk to the server and install a protocol handler that implements the HTTPS protocol. JSSE comes with such a protocol handler.

As an example, here's a simple URL−based client that can retrieve arbitrary URLs:

```
package javasec.samples.ch14;

import java.io.*;
import java.net.*;

public class URLClient {
    public static void main(String[] args) throws Exception {
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection(  );
        BufferedReader br = new BufferedReader(
                   new InputStreamReader(uc.getInputStream(  )));
        String s = br.readLine(  );
        while (s != null) {
            System.out.println(s);
            s = br.readLine(  );
        }
    }
}
```

You can run this code with an HTTP−based URL as follows:

```
piccolo% java javasec.samples.ch14.URLClient http://www.sun.com/
... lots of output from sun.com ...
```

Similarly, by specifying the appropriate property for the HTTPS protocol handler, you can connect to an HTTPS−based URL:

```
piccolo% java \
   −Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol \
   javasec.samples.ch14.URLClient https://www.sun.com/
```

As always, the server (*sun.com* in this case) will present its certificate to the client, which must verify it using its truststore. In this case, we've used the default truststore (*$JREHOME/lib/security/cacerts*), which contains the root certificate of many CAs (including the one used by Sun). If you connect to your own server, you may need to specify the appropriate `trustStore` property.

14.6.1 Verifying HTTPS Hosts

When we wrote our own SSL client socket code, we had to extract the name from the server's certificate and make sure that it represented the host to which we expected to connect. The HTTPS protocol handler will do that for us automatically, and if the hostnames don't match, an `IOException` will be thrown when you attempt to get the input or output stream.

There are times when this verification is insufficient. If you connect to *https://192.18.297.41/*, the default verification will fail. The certificate presented by that site has an embedded name of *www.sun.com*, and even though 192.18.297.41 is the correct IP address, the protocol handler will do a string comparison of 192.18.297.41 and *www.sun.com* and will fail. In cases such as this you may want to look up the IP address of the name in the certificate and see if it matches your target. You may also want to ask the user if it's okay to proceed (regardless of whether the names match).

To handle such situations, you can implement a hostname verifier in order to perform extended hostname verification. Extended hostname verification is used only if the name in the certificate and the hostname in the URL don't match; if the names match, the HTTPS protocol handler does not call the hostname verifier. Hence, a hostname verifier cannot be used to prevent any arbitrary connection.

To use extended verification, write a class that implements the `HostnameVerifier` interface (`com.sun.net.ssl.HostnameVerifier`). This interface contains a single method:

public boolean verify(String urlHostname, String certHostname)
: Verify that the common name from the server certificate should be accepted when attempting to connect to the given URL hostname. Return `true` if you want to accept the connection; `false` otherwise.

Hostname verifiers are installed on instances of the `HttpsURLConnection` class (`com.sun.net.ssl.HttpsURLConnection`), which extends the standard `HttpURLConnection` class. These are returned from the `openConnection( )` method of the URL class.

There are two methods by which you can install a hostname verifier:

public static void setDefaultHostnameVerifier(HostnameVerifier v)
: Set the hostname verifier that will be used by default for all future instances of HTTPS connections.

public void setHostnameVerifier(HostnameVerifier v)
: Set the hostname verifier that will be used for this particular instance of the HTTPS connection. This method must be called before obtaining the input or output socket of the connection.

Here's an example that puts this all together:

```
package javasec.samples.ch14;

import java.io.*;
import java.net.*;

import com.sun.net.ssl.*;

public class HttpsClient {

    static class HttpClientVerifier implements HostnameVerifier {
        public boolean verify(String url, String cert) {
            try {
                // We know the strings don't match, but it's
                // conceivable that their IP addresses do. We
                // could also ask the user here.
                InetAddress iaU = InetAddress.getByName(url);
                InetAddress iaC = InetAddress.getByName(cert);
                return iaU.equals(iaC);
            } catch (Exception e) {
                return false;
            }
        }
    }

    public static void main(String[] args) throws Exception {
        URL u = new URL(args[0]);
        // Alternately, we could set a global verifier like this
        // HttpsURLConnection.setDefaultHostnameVerifier(
        //                  new HttpClientVerifier(  ));

        HttpsURLConnection huc =
                    (HttpsURLConnection) u.openConnection(  );
        // Instead, we set the verifier only for this instance
        huc.setHostnameVerifier(new HttpClientVerifier(  ));

        BufferedReader br = new BufferedReader(
                        new InputStreamReader(huc.getInputStream(  )));
        String s = br.readLine(  );
        while (s != null) {
            System.out.println(s);
            s = br.readLine(  );
        }
    }
}
```

14.6.2 HTTPS Properties

Four properties control how the HTTPS protocol handler operates. We've already seen the first of these, the `java.protocol.handler.pkgs` property that defines which class should be used as the protocol handler. Additionally, you may specify any of the following properties:

https.proxyHost
: If you must go through a proxy server to access the URL in question, set this property to the name of the proxy server.

https.proxyPort
: If you must go through a proxy server that resides on a port other than 80, set this property to the port number.

http.nonProxyHost
: If there are hosts on your local network that should be accessed directly (rather than through the proxy host), set this property to a list of those hosts. The hosts should be separated by the pipe (|) symbol: *www.sun.com*|*www.ora.com*. Note that this property applies to both the HTTP and HTTPS protocols, despite its name (there is no separate `https.nonProxyHost` property).

https.cipherSuites
: If you want to use a particular set of SSL cipher suites, set this property to a list of comma−separated suites that you wish to use.

14.7 Debugging JSSE

Code that is involved in an SSL conversation can be tricky to debug because many of the details that you'd normally handle yourself (key exchange, certificate verification, and so on) are hidden from you. When these operations don't work, it can be difficult to figure out what went wrong.

Complicating this is the fact that setting up an SSL connection is a time−consuming operation. Both parties in the conversation must create a secure random number (an instance of the `SecureRandom` class if they are Java programs). Then the peers must negotiate which key exchange algorithm to use and actually perform the key exchange. Only then is the socket available to send and receive data. So our first tip in working with SSL code is to be patient when you start a program.

Several exceptions are thrown by the JSSE API. These are often self−explanatory. For example, if you attempt to retrieve the certificate chain of a peer from the `SSLSession` object, you will get an `SSLPeerUnverifiedException` if the peer is not verified. However, you will get a

`SocketException` with the somewhat cryptic detail message of "No SSL Sockets" if you specify an incorrect password for a keystore used by an SSL context or an SSL socket factory.

Exceptions are not always thrown when you might expect, however. In particular, an SSL socket will become connected at the socket level even if the SSL protocol negotiation fails. For instance, when an SSL client calls the `createSocket( )` method, it will receive a valid socket even if it is unable to verify the identity of the server to which it is connecting (because, for example, the client used the incorrect truststore). If the client attempts an SSL operation on the socket (such as retrieving the certificate chain in order to verify the server's hostname), an exception will be thrown. If, however, the client just uses the socket, no exception will occur: the `write( )` method will succeed. In this case, the server can read from the socket, but it will get no data. Similarly, the server can write data to the client and the client will see that data was written but be unable to read that data.

In our simple test program, this manifests itself with the following output:

```
piccolo% java SSLSimpleClient localhost 9096
Who is Sylvia?
null
```

Here the client has failed to specify a truststore; it will use the default *cacerts* truststore. Since that truststore does not contain the root certificate that our server uses, the server verification fails. But the client doesn't really know that until it reads data from the server and receives a `null` string. This is another reason why it's important to verify the name of the peer to which you are connecting.

Finally, JSSE specifies a property−based debug facility that supplements the standard debug facility of the `java.security` package. We explained this facility in Chapter 1.

14.8 Summary

In this chapter, we've looked at JSSE and how it provides support for SSL and TLS. SSL is an important protocol because it is so ubiquitous. Since it requires only one participant in a conversation to possess a valid certificate, SSL makes it easy to send and receive secure, encrypted data in an environment with relatively few servers and many clients.

SSL is also the basis of the HTTPS protocol, the secure HTTP protocol. From an application perspective, the HTTPS protocol handler that comes with JSSE makes it simple to talk to HTTPS servers using the same code you'd use to talk to a standard HTTP server.

Chapter 15. Authentication and Authorization

So far, we've examined security mainly from the perspective of how it protects the end user from the outside world. The default sandbox protects end users from writers of malicious Java programs; digital signatures protect the integrity of end user data while encryption protects the confidentiality of end user data.

But how do we protect the rest of the world from end users? That's the topic of this chapter, which focuses on the Java Authentication and Authorization Service (JAAS). JAAS provides a framework through which developers can require users who execute their code to have explicit permission to perform certain operations.

JAAS provides a set of classes that authenticate a user. This typically means that a JAAS−enabled application requires a user to log into it, much like the user logs into his computer (in fact, JAAS often uses the operating system to authenticate the user directly). JAAS also provides a set of classes that authorize users to perform certain operations; this authorization is very similar to the permissions−based authorization that the default sandbox grants to code loaded from particular locations or signed by particular entities.

Like the default sandbox, permissions granted to particular users by JAAS are administered by a system administrator; the system administrator also sets up the default parameters that JAAS uses. However, applications must be modified in order to use JAAS; by default, Java applications do not use the JAAS framework.

In this chapter, we'll look into all aspects of JAAS:

- Administrative steps to enable JAAS
- Programmatic steps to enable JAAS
- Programmatic extensions to JAAS

See Chapter 1, for information on installing JAAS.

15.1 JAAS Overview

JAAS provides a framework based on a pluggable architecture. Like the engines we looked at earlier, JAAS provides a set of abstract classes, and at runtime each program finds the appropriate provider of the necessary class. However, the pluggable architecture is not built into the standard security framework, so we will refrain from referring to the major JAAS classes as engines.

A JAAS−enabled application works like this:

1. The program asks the user to log in, obtaining a user login object.

   Programmatically, this is a simple operation, involving the instantiation of a `LoginContext` object and the invocation of a single method on that object. What happens when that method is invoked can be quite complex and is determined by a system administrator.

   The system administrator is responsible for setting up a file that contains one or more directives indicating what happens when a particular application attempts to log in a user. These directives take the form of login modules that are called to authenticate the user and a series of options that govern how those classes can be used. The classes themselves typically interact with the operating system, using system calls to authenticate the user via Solaris' NIS or NIS+, the Windows NT login service, an LDAP server, or whatever other authentication system is available on the platform.

The system administrator also determines the parameters of the authentication. For example, the user may be required to enter a valid Solaris password. Alternately, she may be required to enter a valid LDAP password, a valid NT password, or a valid password held in a custom database and either a valid Solaris or NT password. One or more of these passwords may be deemed optional. The administrator can set up as few or as many login modules as desired, and any or all of them may be optional or required.

2. The program executes a method call (the `doAs( )` or `doAsPrivileged( )` method), passing in the user login object and the code that should be executed on behalf of that user.

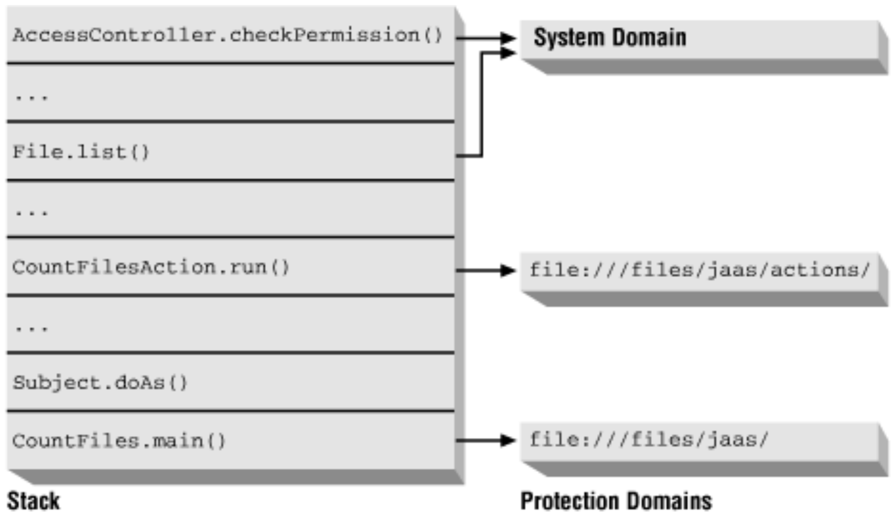
   Programatically, this method is very similar to the `doPrivileged( )` method of the access controller. As we'll see, it accomplishes a similar purpose.
3. Within the context just created, the program executes code that requires a specific permission (e.g., it attempts to count the files in a directory).

   Like all such requests, this code results in a call to the security manager (and hence the access controller) to see if the appropriate permission is granted. As usual, if all the classes on the stack are granted that permission via the standard policy files (or whatever `Policy` class is in effect), the permission will be granted and the call will succeed.

   However, because the call was executed within the context of the `doAs( )` method, the JAAS framework comes into play. JAAS provides a second set of policy files that allow permissions to be granted to code loaded from particular locations and/or signed by particular entities, but only if that code is being executed by a specific authenticated user.

   Figure 15−1 shows how the stack might look when a call is made to the access controller from within the JAAS framework. The application code was initially loaded from *file:///files/jaas/*. That code authenticated the user and passed the authenticated user and an object loaded from *file:///files/jaas/actions/*to the `doAs( )` method. To call the `File.listFiles( )` method, the protection domain *file:///files/jaas/* must have the relevant permissions in the standard Java policy file. The protection domain *file:///files/jaas/actions/* must have those same permissions, but they may be loaded from the standard Java policy file (in which case they apply to all users). Alternately, the permissions may be loaded from a JAAS policy file, in which case they apply only to the user named in that file.

Figure 15−1. The stack and protection domains of a JAAS−enabled application



This last step has an important ramification. Because there are now two sets of policy files, you must be very careful when you set up a program to run with JAAS. The initial code must be granted certain privileges: in addition to the required JAAS−specific permissions, depending on the underlying module used to authenticate the user, other permissions (such as connecting to a nameserver) may come into play. In practice, this means that the setup code runs with a large set of permissions, and it's usually easiest to grant that code all permissions.

The code that runs on behalf of a user, of course, cannot have such broad permissions; we probably don't want it to be able to make arbitrary socket calls or many of the other actions that the setup code performs. To prevent this code from being granted all the permissions of the setup code, you must load it from a different codebase. Hence, a JAAS−enabled program should be partitioned into (at least) two jar files or directories, each of which can be used to specify a different codebase and can therefore have different sets of permissions. Depending on the type of authentication you're performing and the type of activities you want to prevent users from performing, this partitioning is not strictly required, but it's usually necessary.

For developers, then, there are two steps to using JAAS: they must make a call to authenticate the user and execute particular methods on behalf of that user. For administrators, there are three steps: they must configure a set of login modules, configure a set of JAAS policy files, and set up the program's environment correctly.

In the next few sections, we'll look at how all this works with a simple example. We'll start with the programmatic steps required to JAAS−enable code, then we'll look at the administrative steps required to run such code.

15.2 Simple JAAS programming

The JAAS−enabled code is partitioned into two groups: the setup code and the user−specific code.

15.2.1 The JAAS Setup Code

The setup code looks like this:

```
package javasec.samples.ch15;

import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;

public class CountFiles {

    static class NullCallbackHandler implements CallbackHandler {
        public void handle(Callback[] cb) {
            throw new IllegalArgumentException("Not implemented yet");
        }
    }

    static LoginContext lc = null;
    public static void main(String[] args) {
        // use the configured LoginModules for the "CountFiles" entry
        try {
           lc = new LoginContext("CountFiles",
                                new NullCallbackHandler(  ));
        } catch (LoginException le) {
            le.printStackTrace(  );
            System.exit(−1);
        }

        // log in the user
        try {
            lc.login(  );
            // if we return with no exception, authentication succeeded
        } catch (Exception e) {
            System.out.println("Login failed: " + e);
            System.exit(-1);
        }

        // now execute the code as the authenticated user
        Object o =
            Subject.doAs(lc.getSubject(), new CountFilesAction(  ));
        System.out.println("User " + lc.getSubject(  ) + " found " +
                            o + " files.");
        System.exit(0);
    }
}
```

There are three important steps here: first, we construct a `LoginContext` object; second, we use that object to log in a user; and third, we pass that user as one of the parameters to the `doAs(  )` method.

15.2.1.1 The LoginContext class

The first two of these activities are based on the `LoginContext` class (`javax.security.auth.login.LoginContext`). There are four ways to construct a `LoginContext` object:

public LoginContext(String name)
public LoginContext(String name, CallbackHandler cb)
public LoginContext(String name, Subject s)
public LoginContext(String name, Subject s, CallbackHandler cb)

Establish a context by which a user can be authenticated. The details of that authentication are handled by an external configuration file; the name passed to the constructor of this object is referenced from that configuration file.

Certain ways of authenticating users may require callbacks (e.g., to allow the user to type in a password), which we will explore after our simple example. If you use a constructor that does not require a callback and then configure the program to use an authentication method that does require a callback, a `LoginException` is thrown. That exception also is thrown if there is an error in the external configuration files.

The purpose of a login context is to allow retrieval of an authenticated user, which is represented as a subject object. You can pass a preexisting subject to the login context if you want to add credentials to the subject. For now, we'll treat subjects as opaque objects; we'll examine them in detail later in this chapter.

Creating the login context does not authenticate the user; that is done via the `login(  )` method, as our sample code demonstrates. That method is one of three available to this class:

public void login( )

Authenticate the current user and carry out the steps listed in the login configuration file. This method throws a `LoginException` if the login fails.

public void logout( )

Log the current user out. This invalidates the subject object. This method throws a `LoginException` if the logout operation fails.

public Subject getSubject( )

Return the subject object that represents the authenticated user.

15.2.1.2 The Subject class

The `Subject` class (`javax.security.auth.Subject`) is used to represent an authenticated user. In essence, each user is represented as an array of `Principal` objects stored by this class. There is an array of objects because each user is likely to have several identifying characteristics. For example, on my Sun system I have a principal that represents me by username (sdo), one that represents me by user ID (6058), and many that represent me by the groups to which I belong (group 20, group 45). In addition, I may have other identities (such as a database login with a name of scott). Since a principal contains only a single name, my identity is modeled as a set of these principals.

For the most part, subjects are opaque objects. You can retrieve the entire set of principals from the subject object as well as private and public credentials (i.e., keys and certificates) if they are set by the login system. Unless you're implementing your own authentication system, you really use the subject object only as an argument to one of the following static methods of the `Subject` class:

public static Object doAs(Subject s, PrivilegedAction pa)
public static Object doAs(Subject s, PrivilegedExceptionAction pea)
public static Object doAsPrivileged(Subject s, PrivilegedAction pa, AccessControlContext acc)
public static Object doAsPrivileged(Subject s, PrivilegedExceptionAction pa, AccessControlContext acc)

Execute the `run( )` method of the given object on behalf of the given subject (possibly with the given access control context). If the `run( )` method can throw an exception, you must use a method that requires a `PrivilegedExceptionAction` parameter; the exception will be wrapped into a `PrivilegedExecption`.

The `doAs( )` method looks remarkably similar to the `doPrivileged( )` method of the access controller. This is not an accident: the `doAs( )` method sets up special checking that the access controller uses to perform permission checking. The details of how that works are coming up next.

15.2.2 The JAAS User−Specific Code

The code that we'll execute in our simple example looks like this:

```
package javasec.samples.ch15;

import java.io.*;
import java.security.*;

class CountFilesAction implements PrivilegedAction {
    public Object run(  ) {
        File f = new File(File.separatorChar + "files");
        File fArray[] = f.listFiles(  );
        return new Integer(fArray.length);
    }
}
```

From a developer's perspective, nothing special is required to write the user−specific code. In our example, the authenticated user will execute this object, which provides the number of files in the */files* directory.

When the `listFiles( )` method is called, the access controller will be called with the stack shown earlier in Figure 15−1. If this class has the appropriate file permission associated with the current user, it will execute correctly. If not, it will throw a `SecurityException`.

15.3 Simple JAAS Administration

To run our simple example, we must take several administrative steps; in fact, JAAS places a much bigger burden on the administrator than on the developer. The system administrator must configure a set of login modules that will be executed by the login context, write a set of JAAS policy files for the application, and ensure that the program environment is set up correctly to run the application.

15.3.1 Configuring Login Modules

The login context object is quite complex, despite its simple interface. It is built to support a set of pluggable, stackable login modules. A login module is the code that actually authenticates a user. Depending on the module, this may entail either interacting with the user (asking for a login name and password) or using existing information in the user's environment to authenticate the user. A login module may succeed or fail in its attempt to authenticate a user.

Login modules are called pluggable because they are loaded dynamically. Instead of calling specific login modules in your code, the login context looks up the login configuration file to see which classes to call. This allows you to use login modules supplied by third parties.

Login modules are called stackable because you can specify more than one login module in the configuration file. These modules "stack" within the configuration file; they are called in order, and each one can add one or more principal objects to the current subject (i.e., the current user). This is how subject objects end up with multiple principals: they may come from a single login module, or they may come from several login modules.

A sample login configuration file looks like this:

```
CountFiles {
    com.sun.security.auth.module.SolarisLoginModule required;
    com.sun.security.auth.module.JndiLoginModule optional;
};

DBApplication {
    com.sun.security.auth.module.NTLoginModule required;
};
```

The login configuration file can have any arbitrary name. This example contains two tags. The tag of the `CountFiles` entry in the first example is matched to the name that is passed to the login context constructor. Once an entry is found, each of the classes listed is called in order.

The entry for a particular class has this format:

```
classname control−flag [optional parameters];
```

The classname is the full classname of the login module you want to use. The control flag is either `required`, `sufficient`, `requisite`, or `optional`; we'll discuss the meanings of these later. The optional parameters are specified as `name=value` entries. We did not list any in our example, but if you wanted to include a parameter to enable debugging, you'd do it like this:

```
com.sun.security.auth.module.NTLoginModule required debug=true;
```

This prints out certain debugging information on a module−specific basis. All Sun−supplied login modules accept the `debug` flag; other modules accept other parameters as mentioned in their individual documentation.

15.3.1.1 Login control flags

When you stack modules, you can control how they are called via the login control flag. There are four values for this flag:

required
This module is always called, and the user must always pass its authentication test.

sufficient
If the user passes the authentication test of this module, no other modules (except for required ones) will be executed; the user is sufficiently authenticated.

requisite
If the user passes the authentication test of this module, other modules will be executed but (except for required ones) can be failed.

optional
The user is allowed to fail this module. However, if all modules are optional, the user must pass at least one of them.

The idea of stackable modules is crucial to understanding how these flags work because their behavior is altered depending on the order in which they are invoked. Table 15−1 shows how this relationship works. The table assumes that the user has already been successfully authenticated by a module with the flag listed at the top of the column. Then, if a module with the flag listed in the left column of the table is called, the user may fail or must pass the authentication as indicated.

Table 15−1. Behavior of Login Control Flags

	Required	Sufficient	Requisite	Optional
Required	User must pass	User must pass	User must pass	User must pass
Sufficient	User may fail	Not called	User may fail	User may fail
Requisite	User must pass	Not called	User must pass	User may fail
Optional	User may fail	Not called	User may fail	User may fail

This interaction between flags is complicated and is probably best avoided. In fact, because of the way policy files work, it is impossible to take full advantage of mixing the stacked flags. The policy flag lists the class that authenticated the user or the principal of the user (e.g., his username). If you specify one module as sufficient and then a second module as requisite, the entries in the policy file that correspond to the login module listed as requisite will never be granted: the user will never have been logged into that module.

15.3.1.2 Sample login modules

JAAS does not come with any login modules; however, there are three modules available as two separate downloads from Sun's web site. One download contains the Solaris login module and the JNDI login module; the other contains the NT login module and the JNDI login module:

The Solaris login module

The class name of the Solaris login module is `com.sun.security.auth.module.SolarisLoginModule` (that's the name you list in the login configuration file).

This module returns three types of principals:

com.sun.security.auth.SolarisPrincipal
: Contains the login ID (UID) of the user (e.g., sdo).

com.sun.security.auth.SolarisNumericGroupPrincipal
: Contains the group ID (GID) for each group to which the user belongs (e.g., 20). If the user belongs to multiple groups, each group is listed in a separate principal.

com.sun.security.auth.SolarisNumericUserPrincipal
: Contains the user ID (e.g., 6058).

This module requires no interaction with the user; all information is obtained from the user's environment.

The NT login module

The class name of the NT login module is `com.sun.security.auth.module.NTLoginModule`.

This module returns six types of principals:

com.sun.security.auth.NTDomainPrincipal
: Contains a domain name if the user logged into a Windows NT domain, a workgroup name if the user logged into a workgroup, or a machine name if the user logged into a standalone configuration.

com.sun.security.auth.NTUserPrincipal
: Contains the user's NT login name.

com.sun.security.auth.NTSidDomainPrincipal
: Contains the string representation of the Windows NT security identifier (SID) of the domain the user is logged into. If the user is logged into a workgroup or standalone configuration, this principal is not available.

com.sun.security.auth.NTSidGroupPrincipal
: Contains the string representation of the Windows NT SID associated with the group to which the user belongs. If the user belongs to more than one group, each group is listed as a different principal.

com.sun.security.auth.NTSidPrimaryGroupPrincipal
: Contains the string representation of the user's primary group SID.

com.sun.security.auth.NTSidUserPrincipal
: Contains the string representation of the user's SID.

This module requires no interaction with the user; all information is obtained from the user's environment.

The JNDI login module

The JNDI login module allows you to authenticate a user through JNDI (JNDI 1.2 is required). Its class name is `com.sun.security.auth.module.JndiLoginModule`.

This module requires optional parameters in order to know where the JNDI databases are held; those parameters take this form:

```
user.provider.url=name_service_url
group.provider.url=name_service_url
```

The name service URL varies depending on the protocol. For LDAP, you must know the name of the LDAP server and the name of the entry that stores the user or group information. For example, if the server *piccolo* holds user entries in its schema, its URL might be *ldap://piccolo/ou=People,o=Sun,c=US* (depending, of course, on the schema). The schema must be in the format specified by RFC 2307, which means that the user ID will be retrieved where `uid=username`. The password is expected to be in the `userPassword` field.

NIS URLs are specified as *nis://server/domain/user* for users and *nis://server/domain/system/group* for groups (substitute your NIS server and domain name appropriately).

Note that regardless of the protocol, the URLs must be contained in quotes and the last option must be followed by a semicolon:

```
MyJndiApp {
    com.sun.security.auth.module.JndiLoginModule required
        user.provider.url="nis://piccolo/nytech.East.Sun.COM/user"
        group.provider.url=
              "nis://piccolo/nytech.East.Sun.COM/system/group";
}
```

The JNDI login module uses callbacks to obtain the user ID and password (using a technique we'll look at later). The module can be configured to save each ID and password in its shared state so that subsequent attempts to authenticate the user can use the saved data rather than requesting it from the user again. You can control how that state is used by setting any of the following optional arguments to true:

useFirstPass
: When set, retrieve the username and password from the module's shared state. The retrieved values are used for authentication. If authentication fails, no retry is attempted.

tryFirstPass
: When set, retrieve the username and password from the module's shared state. The retrieved values are used for authentication. If authentication fails, ask the user for a new ID and password and authenticate based on those values.

storePass
: When set, store the user ID and password in the shared state only if authentication of the user succeeds and the user ID does not already exist in the shared state.

clearPass
: When set, clear the ID and password in the shared state after authentication is complete.

The principal types returned by this module are identical to those returned by the Solaris login module (e.g., a `SolarisPrincipal`, a `SolarisNumeric−UserPrincipal`, and one or more `SolarisNumericGroupPrincipals`).

Other login modules are certainly possible (such as a Java smart−card module), though none exist at the time of this writing. Check out Sun's Java pages for links to third−party modules.

15.3.2 Writing Policy Files

Once you've written a login configuration file, you must write at least two policy files for the application: a JAAS policy file that grants users particular permissions based on how they were authenticated and a standard Java policy file.

15.3.2.1 Writing JAAS policy files

A JAAS policy file is very similar to a standard policy file: the syntax is almost the same, and the permission types are exactly the same. The only difference is that you must specify the principal type and principal name for each entry. This difference makes it impossible to use `policytool` to edit these policy files; you can create them initially with `policytool` but the final editing must be done by hand. There is no default location for a JAAS policy file; you can give it any name in any directory you choose. Later, we'll show how to pass the name of the JAAS policy file to your application.

The entries in this file will apply to all code executed by the `doAs( )` method we looked at earlier; this file maps the principals associated with the subject passed to the `doAs( )` method with specific permissions. Here's a sample JAAS policy file:

```
grant
    codebase "file:/files/sdo/jaas/actions/"
    signedBy "jra"
    Principal com.sun.security.auth.SolarisPrincipal "sdo" {
        permission java.io.FilePermission "${/}files", "read";
};
```

The `codebase` and `signedBy` entries are exactly the same as we've seen all along, and both are optional. The `Principal` entry is not optional; you must have one or more of them, and each must specify the principal class type that you want to work with, along with the name of the principal that must be authenticated. However, either of them may be an asterisk (*), which means that they match all possible principal classes or all possible principal names. If you use a wildcard, make sure not to enclose it in quotes. It should look like this:

```
grant Principal * * { ... };
```

When matching against a policy file entry that contains multiple principals, the authenticated user must match all principals. The following entry:

```
grant
    Principal com.sun.security.auth.SolarisPrincipal "sdo"
    Principal com.sun.security.auth.SolarisNumericGroupPrincipal "45" {
        permission java.io.FilePermission "${/}files", "read";
};
```

will allow me to read */files* only if my login name is sdo and I am also a member of group 45.

Since my Solaris login is sdo, the first policy file in this section (the one with a codebase) will grant me permission to read the */files* directory when I'm executing code loaded from */files/sdo/jaas/actions*, provided that code is signed by jra. More often, you'll see JAAS policy files that list only the principal (such as the last one we looked at), granting a user permissions regardless of where the code was loaded from or whether it was signed.

15.3.2.2 Writing standard policy files

The setup code for JAAS must have certain permissions. Code that creates a login context and calls the `doAs( )` and `doAsPrivileged( )` methods must have (at least) these permissions:

```
permission javax.security.auth.AuthPermission "createLoginContext";
permission javax.security.auth.AuthPermission "doAs";
permission javax.security.auth.AuthPermission "doAsPrivileged";
```

Of course, this code will need other permissions based on what else it does, as well as all the permissions listed in the JAAS policy file. For simplicity, we generally grant the setup code all permissions.

The code that implements the actual login module must have at least this permission:

```
permission javax.security.auth.AuthPermission "modifyPrincipals";
```

However, different login modules will need different permissions based on what they do. The JNDI login module, for example, needs to be able to open sockets to whatever naming service it is using. Various providers for the JNDI login module need other permissions; for example, the NIS provider needs all permissions (if they aren't granted, it silently fails). If the login modules and other extensions are installed as standard extensions this isn't an issue, as they are given all permissions by the default Java policy files.

15.3.3 Running the Example

Now we'll go through the steps required to run the simple code example shown earlier. We assume for this example that you've downloaded the online code examples into the directory */files* (*C:\ files*) and that your current working directory is */files/javasec/samples/ch15*. If you downloaded the code into a different directory, you'll need to change some of the pathnames in the example. In case you're typing in the code and configuration files yourself, we'll point out the location where each one needs to go.

Here are the steps required to run the example:

1. Partition the code into setup and action code.

   In our case, the setup code is the `CountFiles` class and the action code is the `CountFilesAction` class. The *CountFiles.java* file should be located in the current directory and the *CountFilesAction.java* file should be located in the directory *./actions/javasec/samples/ch15*. This type of partitioning will allow us to create separate policy files for the setup code and the action code.
2. Compile the sample code.

   Since we've segregated the code, we must set the classpath when we compile it. On Solaris, we execute this command:

   ```
   piccolo% javac −classpath ../../..:actions CountFiles.java
   ```

   On Microsoft Windows, the command looks like this:

   ```
   C:\files javac −classpath ..\..\..;actions CountFiles.java
   ```

   Note that this command compiles both source files since the `CountFiles` class references the `CountFilesAction` class; that's why we specified both directories in the classpath.
3. Create the login configuration file.

This entails determining which login module you want to use. In the sample code online, we provide a configuration file that uses a simple login module (named `SimpleLoginModule`) that we write later in this chapter. The configuration file looks like this:

```
CountFiles {
    javasec.samples.ch15.SimpleLoginModule required;
};
```

You can use any other login modules as long as they do not require a callback mechanism to obtain information from the user (our sample program doesn't implement that yet). The Solaris and NT login modules are good candidates to use since they don't require callbacks; we chose to use the `SimpleLoginModule` for this example because it works on all platforms, including Microsoft Windows 95/98. The name and location of this file are arbitrary; in subsequent steps we assume that the file is called *login.conf* and is located in the current directory.

If you're using the simple login module, you'll need to compile it and its associated files:

```
piccolo% javac -classpath ../../.. Simple*.java
```

4. Create the JAAS policy file.

   In the sample code online, *policy.jaas* is such a file. It looks like this:

```
grant codebase "file:///files/javasec/samples/ch15/actions/"
    Principal javasec.samples.ch15.SimplePrincipal "defaultUser" {
    permission java.io.FilePermission "${/}files", "read";
};
```

   If you downloaded the code into a directory other than */files*, you need to change the codebase in this file. The name and location of this file are arbitrary; in subquent steps we assume that the file is called *policy.jaas* and is located in the current directory.

   This file allows the user with the ID defaultUser running classes loaded from the */files/javasec/samples/ch15/actions* codebase to read the */files* directory.
5. Create the standard policy file.

   In the sample code online, *policy* is such a file. It looks like this:

```
grant codebase "file:///files/" {
    permission java.security.AllPermission;
};
```

   As with all policy files, its name and location are arbitrary. This file causes classes loaded from the /files codebase to be given permission to perform any operations. Remember that the codebase doesn't include the package name, so the class file */files/javasec/samples/ch15/CountFiles.class* in the package `javasec.samples.ch15` will be given this permission.
6. To run the program, you must specify the following arguments:

   - An appropriate classpath (like the one we used to compile)
   - `-Djava.security.manager` to enable access checking
   - `-Djava.security.policy` to point to the standard policy file
   - `-Djava.security.auth.policy` to point to the JAAS policy file
   - `-Djava.security.auth.login.config` to point to the login configuration file

   For our example, it gives us this command line on Microsoft Windows systems:

```
C:\files\javasec\samples\ch15> java -classpath ..\..\..;actions \
-Djava.security.manager \
-Djava.security.policy=policy \
-Djava.security.auth.policy=policy.jaas \
-Djava.security.auth.login.config=login.conf \
javasec.samples.ch15.CountFiles
```

For non−Windows systems, replace the semicolon in the classpath argument with the appropriate character (e.g., a colon on Unix systems). You may specify multiple JAAS policy files by using multiple arguments, just as you would with standard policy files.

If all goes well, the program will run to completion.

You should experiment with the configuration files we've just created to see how they work. Try specifying a different username or different permissions, and you'll see different errors. If you have a Solaris or NT system, try specifying one of the platform−specific login modules we listed earlier. Note that you can't use the LDAP login module yet because at this point our sample application doesn't support all the features it needs.

15.4 Advanced JAAS Topics

The simple example we've just shown is enough to get you started with JAAS, but now we'll delve into some optional topics, including how callbacks are used to get information from the user, how to write your own login module, how to deal with permissions that can't be put into a JAAS policy file, and how to use some advanced administration options.

15.4.1 JAAS Callbacks

Login modules such as those for Solaris and NT obtain all their information from the user environment. Other login modules aren't so lucky: they must somehow prompt the user to enter relevant information (such as an ID and password). This is accomplished through the use of JAAS callbacks.

When you construct a login context, you have the option of providing it with an object that implements the `CallbackHandler` interface (`javax.security.auth.callback.CallbackHandler`). This object is sent to the login modules, and if they need information from the user, they use the handler object to obtain it. This in turn is accomplished by using one or more callback objects (`javax.security.auth.callback.Callback`), each of which asks for a certain piece of information (e.g., a password).

If you think your application might ever need to use a login module that requires callbacks, you should register the appropriate handler in your application. Implementing a `CallbackHandler` requires an object that provides this method:

public void handle(Callback[] cb)

Loop through the array of callbacks and provide the information desired by each of them. The application is free to use any method to obtain the appropriate information. If the application does not know how to handle a particular callback, it should throw a `javax.security.auth.callback.UnsupportedCallbackException`; other errors can be encapsulated as an `IOException`.

JAAS provides a series of callback objects, each of which asks for a different piece of information. Theoretically, you can implement the `Callback` interface and create a new type of callback, but such an extension will be proprietary to your application. We'll focus instead on the seven standard callback classes that you may need to handle in your application:

ChoiceCallback
ConfirmationCallback
LocaleCallback
NameCallback
PasswordCallback
TextInputCallback
TextOutputCallback

A class that handles callbacks follows this skeleton:

```
package javasec.samples.ch15;

import java.io.*;
import javax.security.auth.callback.*;

class MyCallbackHandler implements CallbackHandler {
    public void handle(Callback[] cb) throws IOException,
                                      UnsupportedCallbackException {
        for (int i = 0; i < cb.length; i++) {
            if (cb[i] instanceof ...) {
                ... c = (...) cb[i];
                // Set the appropriate data in the Callback object
            }
            else if (cb[i] instanceof ...) {
                // and so on
            } else throw new UnsupportedCallbackException(
                                     cb[i], "MyCallbackHandler");
        }
    }
}
```

You decide which callbacks you need to support. The JNDI login module requires that you handle the name callback and password callback, which is typical. For each type of callback, you'll have an if block, substituting the class name where we have the ellipses.

15.4.1.1 The name callback

The most common callback is the `NameCallback` object (`javax.security.auth.callback.NameCallback`). It provides the following methods:

public String getName( )
: Retrieve the name currently stored in the callback object.

public String getDefaultName( )
: Return the default name stored in the object. The default name is set by the login module when it constructs the callback; it is often `null`. One possible implementation is to use the name of the last user that was authenticated by the module. You are free to present this information to the user or to skip it.

public String getPrompt( )
: Return the prompt stored in the object. The prompt is set by the login module when it constructs the callback.

public void setName(String name)

Store the name in the object. You are responsible for calling this, passing it the appropriate ID for the user.

You can set up nice Swing components to prompt for the name, but here's a simple way to read the name from standard input:

```
if (cb[i] instanceof NameCallback) {
    NameCallback nc = (NameCallback) cb[i];
    System.out.print(nc.getPrompt(  )+ " ");
    System.out.flush(  );
    String name = new BufferedReader
                    (new InputStreamReader(System.in)).readLine(  );
    nc.setName(name);
}
```

15.4.1.2 The password callback

Obtaining a password from the user follows a similar vein: the login module provides a password callback object (`javax.security.auth.callback.Password-Callback`) in which you store the password. These methods are available within the `PasswordCallback` class:

public String getPrompt( )

Return the prompt stored in the object. The prompt is set by the login module when it constructs the callback.

public boolean isEchoOn( )

Echo the password to the user. This is set by the login module when it constructs the password callback. You should check this value and if it is false, you should not echo the password as the user types it in (our sample code doesn't do that simply because you can't turn off the echo of the standard input).

public char[] getPassword( )

Return the password presently stored in the object.

public void clearPassword( )

Clear the password stored in the object.

public void setPassword(char[] pw)

Store the given password in the object.

As is usual with passwords, the API treats them as arrays of characters rather than strings; this allows you to limit the amount of time that they are held in memory by clearing out the array.

To read the password from the standard input, we'll use this code:

```
if (cb[i] instanceof PasswordCallback) {
    PasswordCallback pc = (PasswordCallback) cb[i];
    System.out.print(pc.getPrompt(  )+ " ");
    System.out.flush(  );
    String pw = new BufferedReader
```

```
                        (new InputStreamReader(System.in)).readLine(  );
    pc.setPassword(pw.toCharArray(  ));
    pw = null;   // Let pw be collected as soon as possible
}
```

15.4.1.3 The text input callback

The `TextInputCallback` class (`javax.security.auth.callback.TextInput-Callback`) allows login modules to retrieve arbitrary text from the user. It functions almost exactly like the name callback; only the method names are changed:

public String getText( )
: Retrieve the text currently stored in the callback object.

public String getDefaultText( )
: Return the default text stored in the object. The default text is set by the login module when it constructs the callback; it will often be `null`.

public String getPrompt( )
: Return the prompt stored in the object. The prompt is set by the login module when it constructs the callback.

public void setText(String Text)
: Store the text in the object. You are responsible for calling this, passing it the appropriate string supplied by the user.

15.4.1.4 The text output callback

The `TextOutputCallback` class (`javax.security.auth.callback.TextOutputCallback`) doesn't actually retrieve information from the user; it allows the login module to send a warning or informational message to the user. You can use the methods of this class to specify the message and display it to the user as appropriate:

public String getMessage( )
: Specfiy the message to display to the user.

public int getMessageType( )
: Get the type of the message, which will be one of these final static values: `INFORMATION`, `WARNING`, or `ERROR`.

15.4.1.5 The choice callback

When processing a `ChoiceCallback` object (`javax.security.auth.callback.ChoiceCallback`), you are expected to present a given set of choices to the user and allow her to select one or more of them. You return an array of which choices were selected through this API:

public String getPrompt( )
: Return the prompt to be displayed to the user. The prompt is set by the login module when it constructs the callback.

public String[] getChoices( )
: Return the list of choices that the user should be presented with.

public int getDefaultChoice( )
: Return the index of whichever choice in the array of choices is the default.

public boolean allowMultipleSelections( )
: If this method returns `true`, allow the user to make multiple selections. If it returns `false`, allow only a single selection.

public void setSelectedIndex(int selection)
: Call this to indicate that the given selection in the array of choices has been selected. This method should be used only when the `allowMultiple-Selections( )` method returns `false`.

public void setSelectedIndexes(int[] selections)
: Call this method with an array; each element in the array represents an item that was selected (e.g., if item 3 is selected, put the number 3 into the array; don't set `selections[3]` to some value). If multiple selections are not allowed, this method throws an `UnsupportedOperationException`.

public int[] getSelectedIndexes( )
: Return an array of the currently selected choices. If multiple selections are not allowed, return an array of length 1.

15.4.1.6 The confirmation callback

The `ConfirmationCallback` class (`javax.security.auth.callback.ConfirmationCallback`) allows the login module to ask the user a confirmation question that is answered with a yes/no, yes/no/cancel, ok/cancel, or a similar answer. It has the following API:

public String getPrompt( )
: Return the prompt that should be displayed to the user.

public int getMessageType( )
: Return the message type (`INFORMATION`, `WARNING`, or `ERROR`).

public int getOptionType( )
: The type of callback. This will be either:

YES_NO_OPTION
: Present the user with a yes/no option

YES_NO_CANCEL_OPTION
: Present the user with a yes/no/cancel option

OK_CANCEL_OPTION
: Present the user with an ok/cancel option

UNSPECIFIED_OPTION
: Use the `getOptions( )` method to determine what the options are

public String[] getOptions( )

Retrieve the text for each of the possible options (as well as the number of options, based on the length of the returned array) when the option type is `UNSPECIFIED_OPTION`.

public int getDefaultOption( )

Get the default option. This will be `YES`, `NO`, `OK`, or `CANCEL`, unless the options are unspecified, in which case it will be the index into the array returned by the `getOptions( )` method.

public void setSelectedIndex(int selection)

Set the selected confirmation option. This will be `YES`, `NO`, `OK`, `CANCEL`, or, in the case of unspecified options, the index into the array of options. This is the value you are responsible for setting when you process the callback method.

public int getSelectedIndex( )

Get the currently selected option. This will be `YES`, `NO`, `OK`, `CANCEL`, or the index into the array of unspecified options.

15.4.1.7 The language callback

If the login module needs to determine which locale the user should be authenticated for, it will provide a language callback object. The `LanguageCallback` class has a very simple API:

public Locale getLocale( )

Return the locale currently stored in the object.

public void setLocale(Locale l)

Set the locale in the object. You are responsible for determining the appropriate locale (usually just the current locale) and using this method to set it.

If you know your application will use login modules that do not need callbacks, you can specify `null` as the callback handler (as we did in our first example). If the login modules will use only a few specific callbacks, you can use a limited callback handler. This is particularly relevant for server applications with no user to interact with: you can program name and password callback handlers to return a particular name and password, but you cannot hardwire the arbitrary information for other callbacks. On the other hand, for an interactive application, implementing a callback handler that understands all the possible callbacks gives you the most flexibility.

15.4.2 Writing a Login Module

Say you have a set of IDs and passwords stored in a database somewhere. You could use a proprietary authentication system, but if you're really ambitious you might want to write a login module to authenticate users based on their Java Cards. In this section, we'll show you how to write your own login module.

Our rationale (other than pedagogical) for writing a login module is that we want a simple module that will ease testing and can be used on all platforms. So we're going to write a module that always authenticates a user named "defaultUser." We used this module in our example earlier.

Writing a login module requires implementing the `LoginModule` interface (`javax.security.auth.spi.LoginModule`), which means writing an object with the following methods:

public void initialize(Subject s, CallbackHandler ch, Map sharedState, Map options)

Initialize the login module. The subject should be saved; the login module will store one or more principals in it, perhaps using the callback handler to obtain authentication information. The shared state map can be used to cache results; the options map contains the options read from the login configuration file.

public boolean login( )

Authenticate a user. Information about the user may be obtained from the environment or by using the callbacks. If authentication succeeds, return `true`; otherwise return `false`. If you invoke a callback that throws an exception or otherwise encounter a problem, this method should throw a `LoginException`.

public boolean commit( )

Indicates that this user will be accepted. This method is called only if the user is authenticated by all modules in the login configuration file. At this point, the login module must add the appropriate principal object(s) to the stored subject.

If for some reason the user principal objects cannot be stored, this method should return `false`. Otherwise, it should return `true`. A `LoginException` should be thrown for unrecoverable exceptions.

If this module was configured as optional and it was unable to authenticate the user (i.e., the `login( )` method returned `false`), this method will still be called if other modules authenticated the user. In that case, this method should not store any information into the subject, and it should clean up any saved state.

public boolean abort( )

Indicates that the user will not be accepted. This method is called if the user cannot be authenticated (e.g., a required module failed or no optional module succeeded). The module should clean up any saved state. It may throw a `LoginException` if it encounters an error.

public boolean logout( )

Log the user out; this entails cleaning up any state and removing from the saved subject any principals that this module stored. It may throw a `LoginException` if it encounters an error.

To write a login module, you must have an available class that implements the `Principal` interface. There are several in the JAAS API, although they tend to be login module−specific. We mentioned the Solaris and NT principal classes earlier; the only other available class is the `X500Principal` class (`com.sun.security.auth.X500Principal`). This is an appropriate choice if your login module will be providing X500 names, but in most cases you'll need to write your own principal class. Here's a simple implementation:

```
package javasec.samples.ch15;

import java.security.*;
import java.io.*;

public class SimplePrincipal implements Principal, Serializable {
    private String name;

    public SimplePrincipal(String s) {
        name = s;
    }

    public String getName(  ) {
        return name;
    }

    // Simple Principal objects are equal if they contain the
    // same name.
    public boolean equals(Object o) {
        if (!(o instanceof SimplePrincipal))
            return false;
        return ((SimplePrincipal) o).name.equals(name);
    }
}
```

Note that the principal is serializable. We'll say more about that at the end of the chapter.

Here's the simple login module code:

```
package javasec.samples.ch15;

import java.util.*;
import java.io.IOException;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;

public class SimpleLoginModule implements LoginModule {

    private Subject subject;
    private CallbackHandler callbackHandler;
    private SimplePrincipal principal;
    private boolean debug;

    // State information for the currently authenticated user.
    private String userName = null;
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    public void initialize(Subject s, CallbackHandler cb,
                        Map sharedMap, Map options) {

        subject = s;
        callbackHandler = cb;

        // Initialize any configured options.
        debug = "true".equalsIgnoreCase((String)options.get("debug"));

        // We don't use the shared map to cache results between
        // attempts, but if we did we'd need to save it here.
    }

    public boolean login(  ) throws LoginException {
        if (debug)
            System.err.println("SimpleLoginModule: Login");

        // This is where we'd normally do authentication. If
        // necessary, we could instantiate callback objects
        // and put them in an array and call the callback handler.
        // We could also retrieve information from the sharedMap
        // if we cached a previous login attempt.

        // Normally, we'd set this from the getName(  ) method of
        // the name callback, or from the user environment.
        userName = "defaultUser";

        // We'd set this based on a password match. If we get
        // credentials from the user environment, it will
        // always be true.
        succeeded = true;
        return true;
    }

    public boolean commit(  ) throws LoginException {
        if (debug)
            System.err.println("SimpleLoginModule: Commit");

        if (!succeeded) {
            // We didn't authenticate the user, but someone else did.
            // Clean up our state, but don't add our principal to
            // the subject.
            userName = null;
            return false;
        }

        principal = new SimplePrincipal(userName);
        // defaultUser might already be in the subject if
        // another module authenticated him.
        if (!subject.getPrincipals(  ).contains(principal)) {
            subject.getPrincipals(  ).add(principal);
        }

        // Clean up our internal state.
        userName = null;
        commitSucceeded = true;
        return true;
    }

    public boolean abort(  ) throws LoginException {
        if (debug)
            System.err.println("SimpleLoginModule: Abort");

        if (succeeded == false)
```

```
            // We failed, and so did someone else, so just clean up.
            return false;
        else if (succeeded == true && commitSucceeded == true) {
            // Our login succeeded, but another required module failed.
            // We must remove our principal and clean up.
            logout(  );
        } else  {
            // Our commit failed, even though login succeeded.
            // The rest of our internal state should already
            // have been cleaned up.
            succeeded = false;
        }
        return true;
    }

    public boolean logout(  ) throws LoginException {
        if (debug)
            System.err.println("SimpleLoginModule: Logout");

        subject.getPrincipals(  ).remove(principal);
        principal = null;
        userName = null;
        succeeded = commitSucceeded = false;
        return true;
    }
}
```

Because it always succeeds, this is a good module for testing. However, we've indicated the logic in all methods that is necessary to handle failure modes as well. Remember that since login modules can stack, the subject object may have several principals. This is why we don't add the principal to the subject if it's already there. This is also why we took care to write a good `equals( )` method for the `SimplePrincipal` class; the default comparison of principal objects is almost never correct, since it compares object references rather than the information contained in the object. Also remember that even if our module succeeds, other modules may fail and invalidate the entire login process; this is why it's necessary to keep all the state as we proceed.

15.4.3 The JAAS Policy Class

The implementation of the JAAS policy file is provided by a subclass of the `Policy` class (`javax.security.auth.Policy`). This subclass is similar in construction to the `java.security.Policy` class we examined in Chapter 5, but it is not related to the core `Policy` class. While they provide essentially the same API, they implement two sets of policies and operate independently: one applies to all code, and one applies only to code run under the `doAs( )` method.

The default implementation of the JAAS policy class is provided by the `PolicyFile` class (`com.sun.security.auth.PolicyFile`). This class parses the JAAS policy files and presents the appropriate permissions when asked. If you need a different JAAS policy, however, you can provide a different implementation of the `Policy` class, as we'll show in this section.

One case in which you might want to provide your own JAAS policy class is to implement user−specific permissions based on the principal name of the user. For instance, a file server might have a */files* directory that contains a number of subdirectories, one for each user: */files/sdo*, */files/jra*, */files/fred*, and so on. It is impossible to create a JAAS policy file that allows the subject sdo to read */files/sdo*, the subject jra to read */files/jra*, and so on. However, you can do this by implementing a new policy class.

The JAAS `Policy` class has four key methods:

public static Policy getPolicy( )

Return the currently installed JAAS policy object. You must have the `AuthPermission` named `getPolicy` to invoke this method. This method will always return an object, because there is always a JAAS policy object in effect.

public static void setPolicy(Policy policy)

Set the JAAS policy. You must have the `AuthPermission` named `setPolicy` to invoke this method.

public abstract PermissionCollection getPermissions(Subject subject, CodeSource cs)

Retrieve the permissions that should be granted to the given code loaded from the given code source when it is run by the principal(s) contained in the given subject.

public abstract void refresh( )

Refresh the policies in effect (e.g., by rereading the JAAS policy file).

Except for the subject parameter in the `getPermissions( )` method, this is the same API as the core Java `Policy` class.

Here's how you can use this class to implement user−specific file permissions. We'll assume that there are specific policies in the JAAS policy file that you want to apply, so we won't completely replace the `PolicyFile` class; instead, we'll take the permissions from that class and add to them:

```
package javasec.samples.ch15;

import java.util.*;
import java.io.*;
import java.security.CodeSource;
import java.security.PermissionCollection;
import java.security.Principal;
import javax.security.auth.Subject;
import javax.security.auth.Policy;

public class UserPolicy extends Policy {

    private Policy deferredPolicy;

    public UserPolicy(Policy p) {
        deferredPolicy = p;
    }

    public PermissionCollection getPermissions(Subject s,
                                               CodeSource cs) {
        PermissionCollection pc = deferredPolicy.getPermissions(s, cs);
        if (s == null)
            return pc;      // No subject means no specific permissions
        Set principals = s.getPrincipals(  );
        Iterator i = principals.iterator(  );
        while (i.hasNext(  )) {
            Principal p = (Principal) i.next(  );
            FilePermission fp = new FilePermission(File.separator +
                                "files" + File.separator +
                                p.getName(  ) + File.separator + "−",
                                              "read,write,delete");
            pc.add(fp);
        }
        return pc;
    }

    public void refresh(  ) {
        deferredPolicy.refresh(  );
    }
}
```

This class is constructed with an instance of the policy class; it takes permissions from that class and adds the new file permissions to it. When it is asked for permissions for a particular subject, it gets the standard permissions for that subject, then iterates through the principal names in that subject and adds a file permission for each of them. So the user with the principal name sdo will be allowed to read, write, and delete all files in the entire hierarchy of */files/sdo*.

Depending on the login modules in place, this will grant additional (probably harmless) permissions. When authenticated with the Solaris login module, this will also grant me permissions on */files/45*, */files/6058*, and */files/20* (based on the other principal types in the subject, which contain my login and group ID numbers). If you need to avoid this, base the permission on particular principal types.

To use this, you must instantiate it with an instance of the existing `Policy` class:

```
javax.security.auth.Policy.setPolicy(
              new UserPolicy(javax.security.auth.Policy.getPolicy(  )));
```

This is typically done within the `main(  )` method of the application, but the policy can be changed at any time.

15.4.4 Administering a JAAS Policy

If you write a new JAAS policy class, you can install it either programmatically (as we just showed) or administratively, by placing the following line into the *$JREHOME/lib/security/java.security* file:

```
auth.policy.provider=MyPolicyClass
```

This works only for policy classes that have a default constructor; it won't work for our `UserPolicy` class because that class depends on another `Policy` class, to which it defers most of its work. However, unlike the core `Policy` class, this class need not be on the system classpath to be specified in the *java.security* file.

The standard JAAS policy class will read a policy file specified on the command line as `−Djava.security.auth.policy=policyfile` (as shown earlier). You can also set up the *java.security* file to specify default JAAS policy files by adding this line:

```
auth.policy.url.1=policyURL
```

Substitute the correct URL for `policyURL`. As with standard policy files, you may specify any number of files in the *java.security* file; each must be numbered consecutively, starting with 1. If the command−line `java.security.auth.policy` file is specified with a double equals sign (= =), the entries in the *java.security* file will be ignored. On the other hand, if the property `policy.allowSystemProperty` in the *java.security* file is set to `false`, the command−line property will be ignored.

This is all completely analogous to the core Java policy handling, except that there are no JAAS policy files listed by default in the *java.security* file.

15.4.5 Client/Server Authentication

JAAS is designed to allow you to perform authentication in a client and pass the subject to a server. For that reason, the `Subject` class and all the principal classes used by JAAS are serializable. And your own `Principal` classes should also be serializable.

Using JAAS in a client−server environment poses no special challenges, though it does require some forethought. One scheme is to perform authentication on the client: in the client code, create the `LoginContext` and invoke the `login( )` method on it. Then call the `getSubject( )` method on the context object and send the subject as a serialized object to the server. The server can then use the subject as a parameter to the `doAs( )` method.

The challenge with this method is that you must set up the login configuration file and JAAS environment on the client. If you write your own login module, you must distribute it as well. And you must be careful about environmental issues: if the `Subject` class is loaded on the server from the Java extensions directory (which is what we usually recommend), your own principal classes must also be loaded from that directory by the server, or object deserialization will fail. This is a property of how object serialization handles class loading; if you're not serializing the `Subject` object, it's not a requirement.

The alternate method is to provide authentication on the server. In that case, you must determine how to obtain callback information from the user. If all you need are the user ID and password, the client can simply send that information to the server, which can store it in the necessary callback objects. If you have to handle other callbacks, you must devise your own scheme in order to pass the information between the client and server. Fortunately, you can usually rely only on a user ID and password.

15.4.6 Groups and Roles

JAAS places a large burden on the administrator of a system. One technique that can lessen this burden is to rely on groups or roles. If your system has 1,000 users, you don't want to have 1,000 different entries in the policy file; if you can assemble those users into a few groups, setting up the policy file is much easier.

If your application runs on a platform that already supports groups, this is trivial. On Solaris, you can use the `SolarisNumericGroupPrincipal` class to authenticate users based on the Solaris group to which they belong; you can perform a similar technique using the `NTSidGroupPrincipal` class on NT systems.

If you're writing your own login module, the classes you write to implement the `Principal` interface can also implement the `PrincipalComparator` interface (`com.sun.security.auth.PrincipalComparator`). This allows those classes to implement their own group or role checking.

The `PrincipalComparator` interface contains a single method:

public boolean implies(Subject s)

Determine if the given subject is implied by this principal.

Say that you write a principal class called `DBPrincipal` for database administrators. You could implement the `implies( )` method such that if the name is "DBA," it implies every other `DBPrincipal`:

```
package javasec.samples.ch15;

import java.io.*;
import java.util.*;
import java.security.*;
import javax.security.auth.*;
import com.sun.security.auth.*;

public class DBPrincipal implements Principal,
                         PrincipalComparator, Serializable {
    private String name;

    public DBPrincipal(String name) {
        this.name = name;
    }

    public String getName(  ) {
        return name;
    }

    public boolean implies(Subject s) {
        Set set = s.getPrincipals(DBPrincipal.class);
        Iterator i = set.iterator(  );
        if (i.hasNext(  ) && name.equals("DBA")) {
            // If the subject has any DBPrincipal,
            // they are implied by us (if we're DBA)
            return true;
        }

        // Otherwise, we have to look for an exact match
        try {
            while (true) {
                DBPrincipal p = (DBPrincipal) i.next(  );
                if (p.equals(this))
                    return true;
            }
        } catch (NoSuchElementException nsee) {
            return false;
        }
    }

    public boolean equals(Object o) {
        if (!(o instanceof DBPrincipal))
            return false;
        return ((DBPrincipal) o).name.equals(name);
    }
}
```

15.5 Summary

JAAS allows you to authenticate users and grant them specific permissions based on their identities. Like the rest of the Java security model, it is designed to be very flexible by providing configuration files that determine at runtime how the authentication is performed and which authorizations a particular user should be granted. This moves much of the burden of these tasks from the developer to the system administrator, and it allows policies to be changed at will.

Appendix A. The java.security File

Throughout this book, we've mentioned several modifications that you may make to the file *$JREHOME/lib/security/java.security* in order to modify how various aspects of the Java security policy work. This appendix provides an annotated listing of that file (including entries that are not present in the default version of the file).

```
# The list of security providers (see Chapter 8) that will be consulted for
# all programs. There may be any number of these, as long as they are
# numbered sequentially starting with 1. The order is important, since this
# is the order in which providers will be searched for a particular
# algorithm.
#
# In 1.3, there are two default security providers (the first two in this
# list). JCE and JSSE each provide an additional provider, which we
# configured into this file in Chapter 1.
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
security.provider.3=com.sun.crypto.provider.SunJCE
security.provider.4=com.sun.net.ssl.internal.ssl.Provider

# This is the name of the class that will be used as the Policy object.
# This class must be on the system classpath (e.g., in rt.jar or the
# extensions directory). See Chapter 5.
policy.provider=sun.security.provider.PolicyFile

# This is the name of the class that will be used as the JAAS Policy
# object. This line does not appear in the standard java.security
# file, in which case the class to use is hardwired into JAAS code
# itself. You only put this line in this file to change the class
# to use. Note that, unlike the standard Policy object, this class
# does not have to be on the system classpath. See Chapter 15.
auth.policy.provider=com.sun.security.auth.PolicyFile


# For standard permissions, the default is to have a single system-wide
# policy file and a policy file in the user's home directory.
# You can delete these, or add any additional ones that you like as long
# as they are all numbered consecutively, beginning with 1. See Chapter 2.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy

# There are no default URLs from which to read JAAS policy files.
# You may specify them by uncommenting the next property and specifying
# a valid URL. You may specify any number of these files as well.
# See Chapter 15.
# auth.policy.url.1=someURL

# This is the class used to read the JAAS login configuration file.
# If it is not set, a default, internal implementation is used.
# login.configuration.provider=someClass

# This controls whether or not properties (e.g. ${user.home}) encountered
# within a policy file (either JAAS or standard) are expanded.
# If this is set to false, properties will not be expanded when policy
# files are read. See Chapter 2.
policy.expandProperties=true

# This controls whether or not extra policy files can be specified on the
# command line with -Djava.security.policy=somefile.
# Comment out this line to disable this feature. It applies both to
# standard and JAAS policy files. See Chapter 2.
policy.allowSystemProperty=true

# This controls how Java 2 handles jar files signed by 1.1 utilities.
# When it encounters such a file and this property is false, the
# 1.1 IdentityScope will be used to find the signer of the file;
# if it is found in the scope and is trusted, the code will be
# granted all permissions. See Appendix C.
policy.ignoreIdentityScope=false

# This specifies the default keystore type. For maximum safety, you
# should use jceks. See Chapter 10.
keystore.type=jks

# This is the class that will be instantiated as the system
# IdentityScope (used only in 1.1, though Java 2 programs that
# read 1.1−signed jar files will use it as well).
# See Appendix C.
system.scope=sun.security.provider.IdentityDatabase

# This is the file that the default implementation
# of the IdentityScope class will use to load its database
# NOTE: The syntax of this entry is platform−specific since it is
# the name of a file and not a URL
identity.database=${java.home}/identitydb.obj

# When a class loader does access checking, accessing a class in
# this list of packages will cause an exception to be thrown unless
# all protection domains have a RuntimePermission of
# "accessClassInPackage."+package
# See Chapter 2 and Chapter 4.
package.access=sun.

# If the class loader is performing package definition checks, then
# this is the list of packages in which code cannot define classes
# unless it belongs to a protection domain that has a runtime
# permission of defineClassInPackage.+package. However, no default
# class loader performs that check. See Chapter 2, Chapter 4,
# and Chapter 6. Note that no third−party class is allowed to define
# a class in the java package under any circumstance.
package.definition=

# This property defines the class that will be used by the default
# SSL Server Socket Factory to create SSL server sockets. It is not
# set by default, in which case the class
# com.sun.net.ssl.internal.ssl.SSLServerSocketFactoryImpl will be used.
# Export restrictions prohibit setting this property in the global
# version of JSSE; if you set it in that version, it will be ignored.
# See Chapter 14.
ssl.ServerSocketFactory.provider=\
    com.sun.net.ssl.internal.ssl.SSLServerSocketFactoryImpl

# This property defines the class that will be used by the default
# SSL Socket Factory to create SSL sockets. It is not
# set by default, in which case the class
# com.sun.net.ssl.internal.ssl.SSLSocketFactoryImpl will be used.
# Export restrictions prohibit setting this property in the global
# version of JSSE; if you set it in that version, it will be ignored.
# See Chapter 14.
ssl.SocketFactory.provider=\
    com.sun.net.ssl.internal.ssl.SSLSocketFactoryImpl
```

```
# This property defines the algorithm that the default SSL key
# manager will use. It is not set by default, in which case
# Sun's X509 keymanager will be used. See Chapter 14.
sun.ssl.keymanager.type=SunX509

# This property defines the algorithm that the default SSL trust
# manager will use. It is not set by default, in which case
# Sun's X509 trust manager will be used. See Chapter 14.
sun.ssl.trustmanager.type=SunX509
```

Appendix B. Security Resources

Books are very useful for learning some things, and hopefully you've gotten some benefit from the one you're holding in your hand. However, for some types of information, the Internet remains the better choice. In this appendix, we'll list and discuss various network resources that relate to Java and security.

One reason why this information is better found on the Internet is because it is subject to rapid change. The APIs we've discussed may remain fairly stable (despite the big changes in many of them between 1.1 and Java 2), but the information to be found in these resources is more dynamic.

B.1 Security Bugs

Early in my computer science career, I handed in an exam that ended up receiving a lower grade than I had expected.[A] As part of the exam, I was asked to write an algorithm, prove that it was correct, and then provide an implementation of the algorithm.

[A] Okay, that was not an unusual event for me...

While my algorithm and its accompanying proof were completely correct, my implementation received a failing grade. This was a rather dispiriting result: I had come up with a solution and proved that the solution was correct. But the "real" answer −− the implementation −− was still flawed.

Such is the potential problem with implementing a security model. A lot of design and analysis has gone into Java's default security model, and hopefully you'll put your own effort into making your own applications secure. But no matter how sound the design of a security model, in the end it is the implementation that matters.

In this section, we'll discuss some past bugs in Java's security implementation and list some common resources for finding out about and fixing present bugs.

Few issues in the Java world receive more attention than security bugs; report of a new bug is guaranteed to produce a flurry of activity. As a result, readers of the trade press often have the idea that Java is riddled with security bugs or that it isn't secure to begin with. This is not the case. While some important bugs in Java's security implementation have been reported, the impact of these bugs has (at least until now) been minimal.

Bugs that are reported against Java's security model fall into one of five categories:

1. Reports that are not bugs but that arise from a lack of understanding of Java's security model.

   There are two types of very common bugs in this category: applets that perform annoying tasks and applets that seem to break out of the sandbox. The former category includes applets that take lots of CPU time or otherwise consume many resources. As we mentioned at the outset of this book, such attacks are annoying but are not security attacks.

   The latter category often involves bugs that hinge upon someone having installed a local class file (or worse, a local native library); as we know by now, these local class files are treated as trusted classes. When one of these local classes is able to read (or remove) files on your disk, contact a machine on your local network, or engage in some other potentially malicious behavior, word goes out that Java is not secure or at best has bugs in its security model.

   The lesson to learn from these reports is this: no computer security model is a substitute for vigilant practices by the end user. If your policy is never to run shareware programs downloaded from the Internet, then your policy should be never to install local classes on your system. And while newer versions of browsers, along with the ability in Java 2 to run applications in a secure environment, help to mitigate the potential danger of installing a local class file, such features will never obviate the need for users and system administrators to understand and work with the security model. There may be real bugs in the Java implementation −− but don't assume that all reports you hear about the sandbox being broken fall into that category.

2. Bugs that are misclassified; that is, actual bugs that are reported as being security bugs when they are not.

   As we've seen, security is pervasive in the Java platform −− the bytecode verifier, the class loader, the security manager, and the compiler all have aspects of security to them. Hence, bugs in these areas are often considered security bugs even when they are not. For example, a bug in the bytecode verifier is usually assumed to be a security bug, even if it is not; if the verifier doesn't accept a particular construct that it should accept, for example, no security concerns arise.
3. Web−related bugs that are not Java−specific.

   Often, security problems on the Internet are associated with Java without any direct cause. In particular, bugs related to JavaScript^{TM} and to ActiveX often fall into this category.

   When the first reports of ActiveX security bugs were circulated, there was a lot of discussion about "active content"; the assertion in many quarters was that the security problems that plagued ActiveX were inherent in any active content system. This assertion attempted to place Java in the same light as ActiveX since both were active content systems. The reality is that Java and ActiveX have very different security models.

   Similarly, bugs in JavaScript are often confused with bugs in Java, in part because of the name. It is probably well known by this point, but it doesn't hurt to reiterate: JavaScript and Java are completely different technologies produced by separate companies (AOL and Sun, respectively). The two technologies are complementary in many ways, but they are fundamentally different from a security perspective.

   Finally, Java is not immune to security problems that plague the Web in general. Data that is sent between sites among Java applets and servers can be snooped just like data that is sent via HTTP can be snooped (unless the Java traffic is using SSL or another encryption technique). A hacker that sets up a site to impersonate *XYZ.com* will be able to serve Java applets just as she is able to serve HTML.
4. Bugs in third−party trusted classes.

   When you install third−party classes, it is possible that one of them may breach the security model that you think is in place: it may provide a mechanism for an untrusted class to open a file, for example, based upon the permissions normally given to the third−party class.

   Complicating this factor is the manner in which these classes are often installed: they are often put into a directory and the user's classpath is globally set to include those classes. Now untrusted classes will be able to access the third−party classes.
5. Bugs in the Java implementation.

   There have been several well−publicized bugs that do involve Java's security implementation; and as with any large computer system, there are bound to be others.

This last point should not minimized −− there have been and will be bugs in the Java security implementation. But the potential for bugs and their potential impact must be weighed against the potential benefits of using

Java. I know of one corporation where Java is not allowed to be used for any internal project. This site is not worried about employees doing malicious things to other employees, and they filter out Java class files at their corporate firewall, but developers at this company are still not permitted to use Java for any internal project due to security concerns.

When I asked about this policy, I was told this corporation had "zero−tolerance" for security problems, and the mere risk of a Java security bug was enough for them to forbid the use of Java. Of course, this site that had zero−tolerance for security problems had a floppy disk drive on every one of their desktop computers, and users routinely took files to and from the office via floppy disks. The potential for a virus being spread by floppy disk drive (which is very real) was outweighed for them by the benefit of their users doing work at home. Meanwhile, the thought that Java would somehow spontaneously corrupt their isolated network was, for them, enough to outweigh any of the potential benefits they saw to using Java within their extremely distributed, heterogeneous network. Assessing the security of a platform always involves assessing the potential risks and the potential rewards, though apparently that is sometimes hard to do.

B.1.1 Java Security Bugs

One of the ways to assess the potential impact of Java security bugs is to understand the bugs that have occurred to date and their relative impact. The fact that these bugs have been fairly minor and quickly fixed is of some comfort. That is not to say that a future bug won't be more devastating or harder to fix; the point here is really to shed light on the type of bugs that have been found.

Most of the bugs we'll discuss in this section all have another property: attacks based on these bugs were very hard to construct. In fact, attacks based on these bugs never made it out onto the Internet or other networks; the bugs were all reported by various researchers, and often even the researchers had difficulty in constructing an attack against them. One bug in this list (the Brown Orifice bug) is an exception to that: it was fairly easy to exploit that particular bug (long since fixed, of course).

Here's a chronology of security bugs that have been found in Java through February 2001. There was an additional bug reported in July 1998 regarding the class loader, but this applied only to Netscape's implementation, not to the standard JDK.

DNS spoofing

In February 1996, the first Java security bug was posted. It involved a DNS spoofing scenario in which an applet could make a connection to a third−party host other than the one from which it was loaded. Such an attack required access by the attacker to a DNS server that was used by the user and knowledge of the IP address of the third−party machine. DNS spoofing is a general problem (i.e., this bug falls into category 3 in our list), but Java was fixed in 1.0.1 to circumvent this scenario.

Class loader implementation bug

In March 1996, a bug was found that allowed an applet to load a class referenced by an absolute pathname. This bug was fixed in 1.0.1.

Verifier implementation bug

In March 1996, a bug was discovered that took advantage of an implementation error in the bytecode verifier. An attack via this bug needed to be very sophisticated, but it did allow the applet to perform any operation (delete a file, write a file, etc.) on the user's machine. This bug was fixed in 1.0.2.

URL name resolution attack

In April 1996, a bug related to an obscure network configuration was reported. This bug required that the user's machine be running in a DNS domain that it was not registered to and that the attacker's machine be running in that same DNS domain. This bug was fixed in 1.0.2.

Class loader bug
In May 1996, a bug in the class loader was discovered that allowed two applets loaded in different class loaders to exploit a way of casting between different classes with the same distinct name. This bug was fixed in 1.1.

Verifier implementation bug
In March 1997, Sun discovered a bug in the implementation of the verifier. Exploiting this bug would have required knowledge of the bug itself as well as writing Java bytecodes by hand. This bug was fixed in 1.1.1.

Class signing bug
A bug in the `getSigners( )` method of the `Class` class was discovered in April 1997. This bug allowed code signed by one entity to be treated as if were signed by a different entity (possibly with more access to the user's machine). This bug was fixed in 1.1.2.

Verifier implementation bug
A bug that could allow the VM to crash in the bytecode verifier was discovered in May 1997; this bug was fixed in 1.1.2.

Illegal type casting
A bug related to illegal type casting was reported in June 1996. This bug allowed an applet to undermine the typing system of Java. This bug was fixed in 1.1.3.

Unverified classes
In March and April 1999, two implementation bugs were discovered that allowed code to be run without passing through the bytecode verifier. These bugs were fixed in 1.1.8 and fix was back−ported to 1.1.6_006 and 1.1.7_003.

The Brown Orifice Exploit
This exploit depended upon two different bugs. One was present in the JDK itself; in some circumstances it allowed an applet to accept a connection from a host other than the one from which it originated. That bug was present only in Java 1.1 and was fixed in 1.1.8_005, 1.1.7B_007, and 1.1.6_009. Versions of the Java Plug−in based on 1.1 were also vulnerable; the fix in the JDK applies to the Java Plug−in as well.

Compounding this was a bug in Netscape Navigator 4.x (but not other browsers) that allowed these applets to read files; hence the Brown Orifice Exploit turned a user's machine into a web server, serving all the files on the user's machine. This bug was fixed in Netscape 4.75.

Class loading bug
In November 2000, Sun discovered a potential bug that allows an untrusted class to call into a disallowed class (violating the `accessClassInPackage` directive). This bug is fixed in 1.3.0, 1.2.2_006, 1.2.1_004, 1.1.8_050, 1.1.7B_007, and 1.1.6_009.

Unauthorized command execution
In February 2001, Sun reported a bug that allowed certain untrusted classes to execute arbitrary commands. This bug applied only to classes that had already been granted permission to execute at least one command; in certain circumstances, these classes were able to execute a command other than the one that had been authorized. This bug is fixed in 1.3.0, 1.2.2_007, 1.2.1_004, 1.1.8_006,

1.1.7B_007, and 1.1.6_009.

B.1.2 Tracking Security Bugs

The nature of tracking security bugs makes it impossible to track them through a book such as this; we're sure that the above list is already out of date. Hence, the better way to track security issues with Java's implementation is to check the following resources on the Web periodically.

An important point to realize about these sites and the bugs we've just listed is that much of the research on security implementation bugs occurs outside of Sun. Sun's approach to Java security is to achieve security by openness −− that is, the more people who can examine the platform for implementation bugs, the better that implementation will become. This is one reason why the Java source code is freely available for noncommercial purposes.

http://java.sun.com/sfaq/chronology.html
: This page lists the known bugs in the security implementation (the above list was culled from this page). New bugs and their fixes are reported here first.

http://www.cert.org/
: The CERT organization tracks security−related bugs for all types of computer systems, including Java implementations. Java−related security bugs are often published as CERT advisories.

http://www.cs.princeton.edu/sip/
: Many of the bugs in Java's security implementation have been discovered as a result of work done at Princeton's Security Internet Programming (SIP) group. This page summarizes their work, including several of the bugs that were listed above.

  Work at SIP is funded by many companies, including Sun itself.

news://comp.security.announce
: This newsgroup tracks security−related announcements about all systems, including Java.

http://www.cs.washington.edu/
: The Kimera Project at the University of Washington was responsible for finding some of the bugs that were listed above.

http://www.alw.nih.gov/Security/security−advisories.html
: This site has links to several services that publish advisories when Java (and other) security−related bugs are discovered.

B.2 Third−Party Security Providers

There is an increasing number of third−party security providers for both the standard Java Cryptography Architecture and for the Java Cryptography Extension. A partial list of these security providers follows. Note that many of them are based outside the United States; they arose out of the export restrictions on Sun's implementations that used to be imposed by the U.S. government.

The following list is not exclusive: new providers will certainly have been written in the time this book has been published, and the algorithms provided by each entry in the list are subject to change. In addition to the listed engines, these packages will all provide the necessary key classes and engines to support the algorithms in the package. For more security providers, see http://java.sun.com/products/jce/jce12_providers.html.

- The Cryptix Foundation, LTD (http://www.cryptix.com/)

  The Cryptix−JCE package from The Cryptix Foundation, LTD in the United Kingdom furnishes a security provider that includes implementations of the following engines:

  Message digest:
  Haval, MD2, MD4, MD5, RIPE−MD128, RIPE−MD160, SHA
  Digital signature:
  RSA with MD2, MD4, MD5 and SHA, El Gamal
  Cipher:
  Blowfish, CAST 5, DES, DESede, IDEA, Loki, RC2, RC4, Safer, Speed, Square, El Gamal

  Cryptix is freely available.
- Distributed Systems Technology Centre (http://www.dstc.edu.au/index.html)

  The Java Crypto and Security Implementation (JCSI) of DSTC includes a Java 2 security provider that implements the following algorithms:

  Message digests:
  MD2, MD4, MD5, RIPEMD−160, and SHA
  Digital signatures:
  DSA/SHA, OIE, MD2/RSA, MD5/RSA, RawRSA, RawDSA,
  Cipher:
  Blowfish, DES, DESede, DEA, PBEwithMD5andDES−CBC (PKCS #5), RC2, RC4, RC5, RSA
  Key agreement:
  Diffie−Hellman
  Key Store:
  PKCS12

  JCSI is free for non−commercial use.
- ERACOM Pty Ltd (http://www.eracom.com.au/)

  The JProv provider from ERACOM furnishes a security provider that implements the following:

  Message digest:
  MD2, MD5, SHA
  Digital signature:
  RSA/MD5, RSA/SHA, DSA/SHA
  Cipher:
  DES, DESede, IDEA, CAST, RC2, RC4, RC5
  MAC:
  DES, DESede, IDEA, CAST, RC2, RC5
  Key agreement:
  Diffie−Hellman
- IAIK−JCE (http://jcewww.iaik.at/)

  This package from the Institute for Applied Information Processing and Communications in Austria (IAIK) comes with a security provider that performs the following:

  Digital signatures:
  RSA/MD5 and RSA/SHA
  Message digests:
  MD5 and SHA

Certificate and CRL classes:
: X509

Cipher:
: DES, DESede, IDEA, RC2, RC4

While IAIK must be purchased for commercial use, it is free for noncommercial use.

• RSA Data Security, Inc. (http://www.rsasecurity.com/products/bsafe/cryptoj.html)

The BSafe product suite from RSA Data Security in the United States includes a Java component (Crypto−J, formerly JSafe) that furnishes a security provider with the following algorithms:

Message digest:
: MD5, SHA

Digital signature:
: RSA/MD5, RSA/SHA

Cipher:
: DES, DESede, RC2, RC4, RC5

Key agreement:
: Diffie−Hellman

B.3 Security References

Finally, here is a number of white papers and other references that are of general interest:

http://java.sun.com/security/
: This is the main index site for all security−related features of Java. In particular, this page has links to security white papers, API and tool documentation, security specifications, and more. This site also has links to many of the other sites we've listed here.

http://java.sun.com/sfaq/
: This is the Frequently Asked Questions page for Java security. This page primarily addresses what applets can and cannot do.

http://java.sun.com/j2se/1.3/docs/guide/security/spec/security−spec.doc.html
: This document is the specification for the Java 2 security architecture; it provided invaluable background for this book. When you download the Java 2 documentation, this document can be found at *$JAVAHOME/docs/guide/security/spec/security−spec.html.*

http://www.users.zetnet.co.uk/hopwood/papers/compsec97.html
: This document gives an interesting perspective on the topic of authentication and in particular whether Java's techniques for authentication are secure.

http://www.doc.gov/
: The Department of Commerce of the U.S. government governs and publishes the export restrictions of encryption and can grant exceptions for exporting encryption technology.

http://www.crypto.com/
: The Export Policy Resource page contains a number of links and other references to sites concerned with the U.S. government encryption policies.

Bruce Schneier. Applied Cryptography. John Wiley & Sons, New York, NY. 1996
: Okay, it is not a web site, but this book is another invaluable reference for details of all the cryptographic topics of this book (Mr. Schneier's web site, for the library−impaired, is

http://www.counterpane.com/).

Jonathan Knudsen. Java Cryptography. O'Reilly & Associates, Sebastopol, CA. 1998

For a discussion of implementing cryptographic algorithms in Java with a series of excellent examples, check out this book.

Stephen Thomas. SSL and TLS Essentials. John Wiley & Sons, New York, NY. 2000

If you want to know about all the low−level details of the SSL protocol, this book is an invaluable guide.

Appendix C. Identity−Based Key Management

In Java 1.1, the primary tool that was used for key management was `javakey`, which is based heavily on the `Identity` and `IdentityScope` classes. The `keytool` utility that comes with Java 2 is a better way to implement key management, and the `KeyStore` class on which `keytool` is based is definitely more flexible than the classes on which `javakey` is based. In addition, the javakey database uses some classes and interfaces that have been deprecated in Java 2 −− primarily the `java.security.Certificate` interface.

Nonetheless, for developers who are still using 1.1, a key management system based upon the `Identity` and `IdentityScope` classes is the only possible solution. In this appendix, we'll show how these classes can be used for key management. For each of the techniques discussed in this appendix there is a complementary technique in the `KeyStore` class. In addition, the `Identity` and `IdentityScope` classes have been deprecated in Java 2, so you should really move to the keystore implementation as soon as possible. As we mentioned in Chapter 10, `keytool` can import a javakey−based database.

C.1 Javakey

Administratively, key management in 1.1 is accomplished using `javakey` , which operates on a file that contains public and private keys. Entities in this file that hold private keys are called signers (since they hold the information necessary to create a digital signature); those that contain only a public key are called identities.

The file used by `javakey` is called *identitydb.obj* and is held in the *$JAVAHOME* directory. The location of this file can be changed by setting the property `identity.database` in the *java.security* file, but it cannot be changed on the javakey command line.

C.1.1 Creating Identities and Signers

The first step in operating with `javakey` is to create entries in the javakey database. You must create entries before assigning them keys or certificates. When you create an entry, you can specify whether or not you trust the entry; the `appletviewer`'s security manager allows entries that are marked as trusted to access all resources on the machine.

The options to create entities in the javakey database are:

−c name [true|false]
: Create a new identity (an entry that can hold a public key certificate) with the given name. If you want to trust this identity, specify true; otherwise, specify false (the default).

−cs name [true|false]
: Create a new signer (an entry that can hold a private key and a public key certificate) with the given name. If you want to trust this signer, specify true; otherwise, specify false (the default).

Here's how we create an entry that will eventually hold a private key for signing:

```
piccolo% javakey −cs sdo true
Created identity [Signer]sdo[identitydb.obj][trusted]
```

C.1.2 Generating Keys and Certificates

Once entries in the database are created, you can assign keys and certificates to them. There are two ways to do this: you can generate the credentials, or you can import them. We'll show how to generate the credentials first:

−gk signer algorithm keysize [pubfile] [privfile]
−g signer algorithm keysize [pubfile] [privfile]

Generate a public and private key pair for the given signer using the given algorithm. If the optional files are specified, the encoded public and private keys are saved to those files.

The algorithm name must be DSA, unless you have a third−party security provider that supplies RSA keys, in which case the algorithm name can be RSA. The keysize must match the sizes supported by the algorithm; for DSA it must be between 512 and 1024 and be a multiple of 64.

−gc directivefile

Generate a certificate according to the instructions in the given file. You usually use this command to generate self−signed certificates (as with `keytool`), though you can use this command to generate certificates that are issued by anyone for whom you hold a private key (e.g., you could create a signer for your enterprise and use that signer to issue certificates to all your employees).

The directive is a list of properties (`name=value` pairs); it must include the following:

issuer.name

The name of the entity in the javakey database that will be used to issue the certificate. A private key must already have been generated for this signer.

subject.name

The name of the entity in the javakey database for whom to issue the certificate. The private and public key of this entity must already have been generated.

issuer.cert

If the issuer holds multiple private keys and certificates, this property specifies which of them should be used to sign the certificate. You can use the −l options (see later in this section) to see what number to use here. This property is optional if the certificate being generated is self−signed.

subject.real.name

subject.org.unit

subject.org

subject.country

Specify the components of the X500 name that will be embedded in the certificate.

start.date

end.date

Specify the dates when the issued certificate should be valid. The format of the property must be a format that can be read by the `Date` class (e.g., 03 June 2001).

serial.number

The serial number to assign to the certificate. This number must be unique for each certificate.

signature.algorithm
: The name of the algorithm that should be used to generate the certificate. This property is optional; if it is not specified, a DSA signature will be used. If you specify RSA, you must have the appropriate security provider installed.

out.file
: The name of the file in which an encoded version of the certificate should be saved. This property is optional (the certificate is always saved within the javakey database itself, of course).

If you want to generate a certificate, you must generate the public and private keys first. Hence, you first need to execute:

```
piccolo% javakey −gk sdo DSA 512
Generated DSA keys for sdo (strength: 512).
```

Then you can generate a self−signed certificate:

```
piccolo% javakey −gc sdo_cert
Generated certificate from directive file sdo_cert.
```

Here's what the *sdo_cert* file for this example looks like:

```
issuer.name=sdo
issuer.cert=1
subject.name=sdo
subject.real.name=Scott Oaks
subject.org.unit=JSD
subject.org=Sun Microsystems
subject.country=US
out.file=sdo.x509
start.date=01 Jan 2001
end.date=31 Jan 2001
serial.number=1001
```

Note that the issuer and subject names are the same, which makes it a self−signed certificate.

C.1.3 Exporting and Importing Credentials

Now that we have a certificate, we must export it. In general, you can export it and transmit it to someone else who can import it into his javakey database. This will allow him to run with special permissions code that is signed by you. In theory, you can send the certificate to a CA in order for them to issue an official certificate. However, the CA requires the certificate to be embedded within a certificate signing request, and there is no tool to create such a request in 1.1, so you'd have to write the code to do the conversion yourself.

Here are the commands to export credentials:

−ec idOrSigner certnum certoutfile
: Export certificate number certnum belonging to the given entity to the given file. The certificate will be exported in encoded X.509/DER format.

−ek idOrSigner pubfile [privfile]
: Export the encoded public key (and optionally the encoded private key) of the given entity to the given file(s).

Note that the credentials are exported in binary (DER) format, as opposed to the ASCII−based formats used by `keytool`. This makes the files harder to transmit.

To import credentials, you use one of these commands:

−ik identity keysrcfile
: Import the public key in the given file and associate it with the given identity (which must already exist). The key must be encoded in X.509/DER format.

−ikp signer pubfile privfile
: Import the public and private keys in the given files and associate them with the given signer (which must already exist). The keys must be encoded in X.509/DER format.

−ic idOrSigner certsrcfile
: Import the certificate in the given file and associate it with the given entity. If the entity already has a public key, the public key must match the key held in the certificate.

The credentials that are to be imported must also be encoded in DER format. Hence, if you get a certificate issued by a CA, you must decode that certificate (again in an ASCII format) into a DER format before you import it into the javakey database. Hence, these commands are generally used to import files that were previously exported by `javakey`.

C.1.4 Signing a jar File

In 1.1, `javakey` is also used to sign a jar file. To sign a jar file, you must have an entity in the database that holds a private key and a certificate; that entity is used in conjunction with this command:

−gs directivefile jarfile
: Sign the jar file according to the specifications in the given file. The directive file is a properties file that specifies the following properties:

  signer
  : The entity in the database to use to sign the file.

  cert
  : The certificate number to use when signing the file.

  file
  : The base name to use when creating the signature file. For example, if this property is SDO, the signature and block files will be SDO.SF and SDO.DSA.

  out.file
  : The name of the signed jar file. This is optional; if you use it, the signature information is added to the input jar file.

C.1.5 Miscellaneous javakey Commands

`javakey` also supports the following miscellaneous commands:

−t idOrSigner [true|false]
: Assign the given trust level to the given entity.

−l
: List the names of all entities in the database.

−ld
List and provide detailed information about all entities in the database.

−li idOrSigner
Provide detailed information about the specified entity.

−r idOrSigner
Remove the given entity from the database.

−ii idOrSigner
Set an information string for the given entity. The information string is printed with the detailed information for an entity.

−dc certfile
Display the certificates stored in the given file (this command does not use the javakey database).

C.2 Identities

Now we'll turn to the programmatic support for key management in Java 1.1, which is based on a set of classes that deal with the notion of identity: the entity to which a key belongs. An identity can represent an individual or a corporation (or anything else that can possess a public and a private key). Key management in 1.1 is only concerned with managing public and private keys; none of these classes understand the notion of a secret key.

C.2.1 The Identity Class

First we'll look at the primary class used to encapsulate an entity that has a public key, the `Identity` class (`java.security.Identity`):

public abstract class Identity implements Principal, Serializable
Implement an identity −− an entity that has a public key. Although it is an abstract class, it contains no abstract methods.

An identity object holds only a public key; private keys are held in a different type of object (the signer object, which we'll look at a little later). Hence, identity objects represent the entities in the world who have sent you their public keys in order for you to verify their identity.

An identity contains five pieces of information:

- A name −− the name of the identity; this satisfies the `Principal` interface that the identity implements.
- A public key.
- An optional information string describing the identity.
- An optional identity scope to which it belongs. Identities can be aggregated into a collection, which is called an identity scope.
- A list of certificates that vouch for the identity.

Note that the default implementation of an identity object carries with it no notion of trustworthiness. You're free to add that feature to your own identity class.

C.2.1.1 Using the identity class

If you want to use an identity object, the following methods are at your disposal:

public final String getName( )
: Return the name of the identity.

public final IdentityScope getScope( )
: Return the identity scope to which the identity belongs.

public PublicKey getPublicKey( )
: Return the public key associated with the identity.

public void setPublicKey(PublicKey key) throws KeyManagementException
: Set the public key associated with the identity to the given public key. This replaces any previous public key as well as any previous certificates associated with this identity. If the public key is already associated with another identity in the identity scope to which this identity belongs, a `KeyManagementException` is thrown. The implementation of this method in the base class does not actually check the identity scope to see if the key already exists in another identity; it's up to the concrete subclass to provide this functionality.

public String getInfo( )
: Return the information string associated with the identity.

public void setInfo(String info)
: Set the information string in the identity, which replaces any existing information string.

public void addCertificate(java.security.Certificate certificate)
: Add the given certificate to the list of certificates in the identity. If the identity has a public key and that public key does not match the public key in the certificate, a `KeyManagementException` is thrown. If the identity does not have a public key, the public key in the certificate becomes the public key for the identity. Like the `setPublicKey( )` method, this should generate a `KeyManagementException` if this conflicts with another key in the identity scope, but the implementation in the base class doesn't automatically provide that.

public void removeCertificate(java.security.Certificate certificate)
: Remove the given certificate from the list of certificates in the identity. If the given certificate isn't in the identity's list of certificates, no exception is thrown.

public java.security.Certificate[] certificates( )
: Return a copy of the array of certificates held in the identity. The array itself is a copy of what is held by the object, but the certificate objects themselves are not.

public final boolean equals(Object id)

Test if the given identity is equal to the current object. Identities are considered equal if they are in the same scope and have the same name. Otherwise, they are considered equal if the `identityEquals( )` method returns `true`. By default, identities in different scopes are considered equal by the `identityEquals( )` method if they have the same name and the same public key.

There are two ways to obtain an identity object −− via the `getIdentity( )` method of the `IdentityScope` class or by implementing and constructing an instance of your own subclass of the `Identity` class.

C.2.1.2 Implementing an Identity class

An application that wants to work with identities will typically provide its own identity class. A typical implementation of the `Identity` class is trivial:

```
package javasec.samples.appc;

import java.security.*;

public class SimpleIdentity extends Identity {
    public SimpleIdentity(String name) throws KeyManagementException {
        super(name);
    }
}
```

Because all of the methods in the `Identity` class are fully implemented, our class need only construct itself. Here are the constructors in the `Identity` class that we have the option of calling:

protected Identity( )
: Construct an unnamed identity. This constructor is not designed to be used directly; it is provided for use by object serialization only.

public Identity(String name)
: Construct an identity object that does not belong to an identity scope.

public Identity(String name, IdentityScope scope) throws KeyManagementException
: Construct an identity object that belongs to the given scope. A `KeyManagementException` is thrown if the given name already exists in the identity scope.

We've chosen in this example only to implement the second of these constructors.

Other than the constructor, we are not required to implement any methods in our class. If you are implementing an identity within an identity scope, there are methods that you'll need to override in order to get the expected semantics.

Our identity class has one other option available to it, and that is the ability to determine when two identities will compare as equal (via the `equals( )` method). The `equals( )` method itself is `final`, and it will claim that two identities are equal if they exist in the same scope and have the same name. If either of those tests fails, however, the `equals( )` method relies on the following method to check for equality:

protected boolean identityEquals(Identity id)
: Test for equality between the given identity and this identity. The default behavior for this method is to return `true` if the identities have the same name and the same key.

If your identity class has other information, you may want to override this method to take that other information into account.

C.2.1.3 The Identity class and the security manager

The `Identity` class uses the `checkSecurityAccess( )` method of the security manager to prevent many of its operations from being performed by untrusted classes. Table C−1 lists the methods of the `Identity` class that make this check and the argument they pass to the `checkSecurityAccess( )` method.

Table C−1. Methods in the Identity Class that Call the Security Manager

Method	Argument
`setPublicKey( )`	`set.public.key`
`setInfo( )`	`set.info`
`addCertificate( )`	`add.certificate`
`removeCertificate( )`	`remove.certificate`
`toString( )`	`print`

The argument to the `checkSecurityAccess( )` method is constructed from four pieces of information: the name of the class that is providing the implementation of the identity class, the string listed in the table above, the name of the particular identity in question (that is, the string returned by the `getName( )` method), and the name of the class that implements the identity scope to which the identity belongs (if any).

In common implementations of the security manager, this string is ignored and trusted classes are typically able to work with identities while untrusted classes are not.

C.2.2 Signers

An identity has a public key, which can be used to verify the digital signature of something signed by the identity. In order to create a digital signature, we need a private key. An identity that carries with it a private key is modeled by the `Signer` class (`java.security.Signer`):

public abstract class Signer extends Identity

A class to model an entity that has both a public key and a private key. Since this is a subclass of the `Identity` class, the public key comes from the implementation of that class, and a signer class needs only to be concerned with the private key.

The `Signer` class is fully implemented even though it is declared as abstract; an implementation of the `Signer` class need not implement any methods.

C.2.2.1 Using the Signer class

A signer is used just like an identity, with these additional methods:

public PrivateKey getPrivateKey( )

Return the private key of the signer.

public final void setKeyPair(KeyPair pair)

Set both the public and private key of the signer. Since public and private keys must match in order to be used, this class requires that in order to set the private key, the public key must be set at the same time. If only one key is present in the key pair, an `InvalidParameterException` is thrown. The act of setting the public key might generate a `KeyManagementException` (a subclass of `KeyException`, which this method throws).

Except for these two operations, a signer is identical to an identity.

C.2.2.2 Implementing a signer

Signers are trivial to implement, given that none of their methods are abstract. Hence, it is simply a matter of calling the appropriate constructor:

```
package javasec.samples.appc;

import java.security.*;

public class SimpleSigner extends Signer {
    public SimpleSigner(String name) throws KeyManagementException {
        super(name);
    }
}
```

Note an unfortunate problem here: if you've added additional logic to your identity subclass, your signer subclass cannot use that logic. Your own signer subclass must extend Java's `Signer` class, not your own identity subclass.

C.2.2.3 Signers and the security manager

In addition to the security checks that will be made as part of the methods of the `Identity` class, the signer class calls the `checkSecurityAccess( )` method of the security manager in the following cases with the strings in Table C−2.

Table C−2. Methods of the Signer Class that Call the Security Manager

Method	Parameter
`getPrivateKey( )`	`get.private.key`
`setKeyPair( )`	`set.private.keypair`

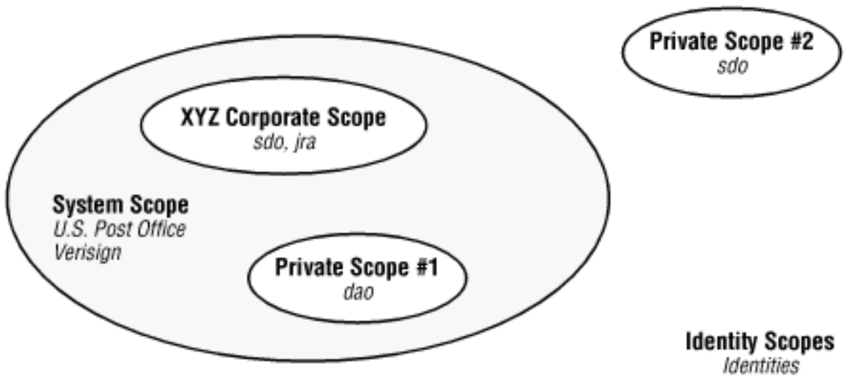
As with the `Identity` class, the actual string passed to the security manager is preceded with the name of the class, and the name of the identity is appended to the class along with the name of the identity's scope.

C.3 Identity Scopes

The database that an identity is held in is an identity scope. There can be multiple identity scopes in a Java program, though typically there is only a system identity scope. By default, the system identity scope for all Java programs is read from a file; this file is the database that `javakey` operates on. But the architecture of an identity scope can be more complex than a single scope.

As Figure C−1 shows, multiple identity scopes can be nested, or they can be disjoint. This is because an identity scope may itself be scoped −− that is, just like an identity can belong to a particular scope, an identity scope can belong to another scope.

Figure C−1. Identity scopes



This architecture is not as useful as it might seem since the identity scope class does not give any particular semantics to the notion of a nested identity scope. If you search the system scope in the figure for `sdo`'s identity, you may or may not find it, depending on how the system identity scope is implemented. That's because there's no requirement that an identity scope recursively search its enclosed scopes for any information. And the default identity scope does not do such a recursive search.

This is not to prevent you from writing identity scope classes that use such semantics −− indeed, writing such a scope is the goal of this appendix.

The idea of an identity scope, of course, is to hold one or more unique identities. However, possible implementations of an `IdentityScope` class (`java.security.IdentityScope`) are conceivably more complicated than that because of the definition of this class:

public abstract class IdentityScope extends Identity

Implementations of this class are responsible for storing a set of identities and for performing certain operations on those identities.

Hence, an identity scope is also an identity. That means that an identity scope might have a name and a public key, which gives you the ability to model an identity database in very different ways. Conceivably, you might want an identity scope for an organization that contains all the identities of individuals within that organization. Rather than having a separate identity for the organization itself, the organization's identity can be subsumed by the identity scope. Since the organization itself also needs a name and a public key, this type of model might offer some flexibility over the alternative: a model that just has a list of identities, some of which are individuals and one of which is the organization.

However, we'll ignore that possibility for now, and just explore the identity scope class with a view to its simplest use: as a holder of one or more identities.

C.3.1 Using the IdentityScope Class

The `IdentityScope` class is an abstract class, and there are no classes in the core API that extend the `IdentityScope` class. Like other classes in the security package, instances of it may be retrieved by a static method (albeit with a different name than we've been led to expect):

public static IdentityScope getSystemScope( )

Return the default identity scope provided by the virtual machine. For `javakey`, this is the identity scope held in the *identitydb.obj* file in the user's home directory (or an alternate file specified in the *java.security* property file).

Once you have retrieved the system's default scope (or any other identity scope), you can operate on it with the following methods:

public abstract int size( )

Return the number of identities that are held in this scope. By default, this does not include the number of nested identities in other scopes that are held in this scope.

public abstract Identity getIdentity(String name)

Return the identity object associated with the corresponding name.

public abstract Identity getIdentity(Principal principal)

Using the principal's name, return the identity object associated with the corresponding principal.

public abstract Identity getIdentity(PublicKey key)

Return the identity object associated with the corresponding public key.

public abstract void addIdentity(Identity identity)

Add the given identity to this identity scope. A `KeyManagementException` is thrown if the identity has the same name or public key as another identity in this scope.

public abstract void removeIdentity(Identity identity)

Remove the given identity from this identity scope. A `KeyManagementException` is thrown if the identity is not present in this scope.

public abstract Enumeration identities( )

Return an enumeration of all the identities in this scope.

For the most part, using these methods is straightforward. For example, to list all the identities in the default identity database, we need only find the system identity scope and enumerate it:

```
package javasec.samples.appc;

import java.security.*;
import java.util.*;

public class ListIdentityScope {
    public static void main(String args[]) {
        try {
            IdentityScope is = IdentityScope.getSystemScope(  );
            System.out.println(is);
            Enumeration e = is.identities(  );
            while (e.hasMoreElements(  )) {
                Identity id = (Identity) e.nextElement(  );
                System.out.println(id);
            }
        } catch (Exception ex) {}
    }
}
```

There is one exception to this idea of simplicity, however. An identity scope is typically persistent −− the `javakey` database is in a local persistent file, and you could write your own scope that was saved in a file, a database, or some other storage. However, you'll notice that there are no methods in the `IdentityScope` class that allow you to save the database for a particular scope. Hence, we could add a new identity to the system identity scope like this:

```
IdentityScope is = IdentityScope.getSystemScope(  );
Identity me = somehowCreateIdentity("sdo");
try {
        is.addIdentity(me);
} catch (KeyManagementException kme) {}
```

That adds an `sdo` identity to the system identity scope for the current execution of the virtual machine, but unless we can somehow save that scope to the *identitydb.obj* file, the `sdo` identity will be lost when we exit the virtual machine. Unfortunately, there are no public methods to save the identity scope.

As an aside, we'll note that the *identitydb.obj* file just happens to be the serialized version of an `IdentityScope` object −− to save the database, we need only open an `ObjectOutputStream` and write the `is` instance variable to that output stream.

There's another point here that we must mention: Java's notion of the system identity scope expects to hold identity objects that are instances of a particular class that exists only in the `sun` package. This means that we can't actually write a fully correct `somehowCreateIdentity( )` method −− we can create identities, but they will not be of the exact class that the system identity scope expects. This can affect some of the operations of the javakey database since some of those operations are dependent on properties of the Sun implementation of an identity that are not in the generic idea of an identity. When we write our own identity−based database at the end of this appendix, that will no longer be a problem (but we won't be able to use `javakey` on that database, either).

C.3.2 Writing an Identity Scope

We'll now implement our own identity scope, which will be one of the classes that we'll use at the end of this appendix to put together an identity−based key management database. We'll write a generic identity scope that implements the notion that its identities are held in a file:

```
package javasec.samples.appc;

import java.io.*;
import java.security.*;
import java.util.*;

public class XYZFileScope extends IdentityScope {
    private Hashtable ids;
    private static String fname;

    public XYZFileScope(String fname) throws KeyManagementException {
        super("XYZFileScope");
        this.fname = fname;
        try {
            FileInputStream fis = new FileInputStream(fname);
            ObjectInputStream ois = new ObjectInputStream(fis);
            ids = (Hashtable) ois.readObject(  );
        } catch (FileNotFoundException fnfe) {
            ids = new Hashtable(  );
        } catch (Exception e) {
            throw new KeyManagementException(
                        "Can't load identity database " + fname);
        }
    }

    public int size(  ) {
        return ids.size(  );
    }

    public Identity getIdentity(String name) {
        Identity id;
        id = (Identity) ids.get(name);
        return id;
    }

    public Identity getIdentity(PublicKey key) {
        if (key == null)
            return null;
        Identity id;
        for (Enumeration e = ids.elements(); e.hasMoreElements(  ); ) {
            id = (Identity) e.nextElement(  );
            PublicKey k = id.getPublicKey(  );
            if (k != null && k.equals(key))
                return id;
        }
        return null;
    }

    public void addIdentity(Identity identity)
                            throws KeyManagementException {
        String name = identity.getName(  );
        if (getIdentity(name) != null)
            throw new KeyManagementException(
                            name + " already in identity scope");

        PublicKey k = identity.getPublicKey(  );
        if (getIdentity(k) != null)
            throw new KeyManagementException(
                            name + " already in identity scope");
        ids.put(name, identity);
    }

    public void removeIdentity(Identity identity)
                                throws KeyManagementException {
        String name = identity.getName(  );
        if (ids.get(name) == null)
            throw new KeyManagementException(
                            name + " isn't in the identity scope");
        ids.remove(name);
    }

    public Enumeration identities(  ) {
        return ids.elements(  );
    }

    public void save(  ) {
        try {
            FileOutputStream fos = new FileOutputStream(fname);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(ids);
        } catch (Exception e) {
            System.out.println(e);
            throw new RuntimeException("Can't save id database");
        }
    }
}
```

Let's delve into the implementation of this class. First, there are two instance variables. The `ids` variable will hold the identities themselves; we've decided to hold the identities in a hashtable so that we can easily search them based on a key. That key will be their name, which makes locating identities in this scope by name very easy (but notice that locating them by public key is harder). The second variable, `fname`, is the name of the file that will hold the persistent copy of this identity scope.

There are three constructors in the `IdentityScope` class that are available to us:

protected IdentityScope( )

Construct an unnamed identity scope. This constructor is not designed to be used by programmers; it is provided only so that an identity scope may be subject to object serialization.

public IdentityScope(String name)

public IdentityScope(String name, IdentityScope scope)

Construct an identity scope with the given name. If an identity scope is specified, the new identity scope will be scoped within the specified scope; otherwise, the new identity scope will have no scope associated with it (like Private Scope #2 in figure Figure C−1). A `KeyManagementException` will be thrown if an identity or identity scope with the desired name already exists in the given scope.

In our case, we've chosen only to provide our identity scope with a name. After calling the appropriate superclass constructor, our class opens up the stored version of the identity database and reads it in. Like the default `javakey` implementation, we've chosen the simple expedient of object serialization to a persistent file to provide our storage. If the file isn't found, we create an empty identity scope.

We've provided a simple `save( )` method that serializes the private database out to the same file that we read it in from; this method has a package protection so that it will only be accessible by the code we develop. The remaining methods in our class are all methods we are required to implement because they are methods that are abstract in our superclass. Because we're storing identities in a hashtable, their implementations are usually simple:

- The `size( )` method can simply return the size of the hashtable.
- The `getIdentity(name)` method can simply use the name as the lookup key into the hashtable.
- The `getIdentity(key)` method is the most complex method, although only slightly: it merely needs to enumerate the identities and test each one individually to see if the keys match.
- The `addIdentity( )` method can search to make sure that the name and public key of the new identity are unique and then simply store the identity into the hashtable with the name as its key.
- The `removeIdentity( )` method can just tell the hashtable to remove the identity with the appropriate key.
- The `identities( )` method can just return the hashtable enumeration.

There is one remaining protected method of the `IdentityScope` class:

protected static void setSystemScope(IdentityScope scope)

Set the system identity scope to be the given scope.

We haven't used this method in this example, but it is one that we'll rely on later when we extend this example. This method replaces the system identity database. Replacing the system database makes things easier for developers. When developers need to operate on identities, they expect to access those identities through the system database. Now that our class is the system database, we can return identities whether they exist in the user's private key database or in the shared public key database.

C.3.3 IdentityScope and the Security Manager

Like the `Identity` class, the `IdentityScope` class uses the `checkSecurity−Access( )` method of the security manager to protect many of its operations from being performed by untrusted classes. This method is called by the `setSystemScope( )` method (with an argument of `set.system.scope`); no other methods of the `IdentityScope` class call this method by default.

However, in the default identity scope implemented in the `sun` package, in the following situations, these methods call the `checkSecurityAccess( )` method with the given string:

- When the `getIdentity( )` method would return a signer −− that is, an identity that has a private key ("get.signer")
- When the `addIdentity( )` and `removeIdentity( )` methods are called ("add.identity" and "remove.identity", respectively)

- When the database is written to a file via object serialization ("serialize.identity.database")

When we implemented the abstract methods of our `IdentityScope` class, we could have made the decision to let the security manager override the ability of an untrusted (or other) class to perform these operations. Hence, a more secure implementation of the `getIdentity( )` method might be:

```
public Identity getIdentity(String name) {
    Identity id;
    id = (Identity) ids.get(name);
    if (id instanceof Signer) {
        SecurityManager sec = System.getSecurityManager(  );
        if (sec != null)
            sec.checkSecurityAccess("get.signer");
    }
    return id;

}
```

C.4 Key Management in an Identity Scope

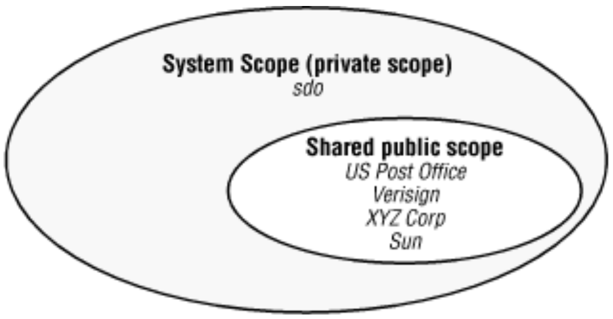
We're now going to put together the identity scope with the information about the identity class to produce another key management system. One of the primary limitations of the default identity scope is that it's based upon a single file. If you're in a corporation, you may want to have an identity scope that encompasses the public keys of every employee in the corporation −− but you can't afford to put the private keys of the employees in that database. Every employee needs read access to the database to obtain his or her own key; there's no practical way with a single identity scope to prevent these users from reading each other's private keys.

Hence, in this example, we're going to develop an identity scope that provides for the architecture shown in Figure C−2.

There are two simple goals to this example:

- There should be a central database (identity scope) managed by the system administrators of the XYZ Corporation. This database will hold the public keys of all identities that are used in the system, along with a security level that is assigned to each identity.
- Each user should have a private database that holds the user's private key. The user's private key will be certified by the XYZ Corporation itself, so this private database will need to have the public key of the XYZ Corporation. We'll make this scope the system scope so that it can encapsulate the knowledge that there are two scopes in use; to a program, it will appear as only a single scope.

Figure C−2. A key management architecture



This architecture allows a program to access the user's private key but not anyone else's private key; it also allows the corporation to set security policies for classes that are signed by particular entities.

There's a certain schizophrenic approach that a system administrator must take in order to use a system like the one we're describing here. Many of the operations that are provided by `javakey` cannot be duplicated by a standard Java program. Hence, we must always rely on `javakey` to perform certain operations (like importing a 1.1−based certificate), and then we need to convert from the javakey database to our own database.

We must implement three classes for this example: an identity class, a signer class, and a shared identity scope class (which will be based upon the `XYZFileScope` class that we showed above).

C.4.1 Implementing an Identity Class

First, let's look at an implementation of the `Identity` class:

```
package javasec.samples.appc;

import java.security.*;

public class XYZIdentity extends Identity {
    private int trustLevel;

    protected XYZIdentity(  ) {
    }

    public XYZIdentity(String name, IdentityScope scope)
                            throws KeyManagementException {
        super(name, scope);
        scope.addIdentity(this);
        trustLevel = 0;
    }

    public void setPublicKey(PublicKey key)
                            throws KeyManagementException {
        IdentityScope is = getScope(  );
        Identity i = is.getIdentity(key);
        if (i != null && !equals(i))
            throw new KeyManagementException("Duplicate public key");
        super.setPublicKey(key);
    }

    public void addCertificate(Certificate cert)
                                throws KeyManagementException {
        Identity i = getScope().getIdentity(cert.getPublicKey(  ));
        if (i != null && !equals(i))
            throw new KeyManagementException("Duplicate public key");
        super.addCertificate(cert);
    }

    public int getTrust(  ) {
        return trustLevel;
    }

    void setTrust(int x) {
        if (x < 0 || x > 10)
            throw new IllegalArgumentException("Invalid trust level");
        trustLevel = x;
    }
    public String toString(  ) {
        return super.toString(  ) + " trust level: " + trustLevel;
    }
}
```

We've chosen in this class to ensure that an identity always belongs to a scope and so we only provided one constructor. There's a somewhat confusing point here, however. Constructing an identity as part of a scope does not automatically add that identity to the scope. That logic is required either in the constructor (as we have done), or the design of the class will require that the developer using the class explicitly assigns the identity to the scope later. The former case is probably more useful.

Other than the constructor, we're not required to implement any other methods in our identity class. However, we've chosen to override the `setPublicKey( )` and `addCertificate( )` methods so that those methods throw an exception when an identity is to be assigned a public key that already exists in the identity scope. You'll recall that when we first introduced the `Identity` class, we mentioned that this logic was not present. Adding that logic is a simple matter of checking to see if the public key in question is already in the identity scope.

Finally, we've introduced a variable in our identity to determine the level of trust that we place in this identity. This is similar to the binary option that `javakey` gives us as to whether an identity is trusted or not; in our version, we allow the identity to have a level of trust. A trust level of 3 might indicate that the identity is fully trusted and hence should have access to all files; a level of 2 might indicate that the identity should be allowed access only to files in the user's temporary directory; a level of 1 might indicate that the identity should never be allowed to access a local file. The point is, the notion of trust associated with an identity is completely up to the programmer to decide −− you're free to assign whatever semantics you like for this (or any other value) or to dispense with such an idea altogether. The idea behind this variable is that the security manager might use it (or other such information) to determine an appropriate security policy.

C.4.2 Implementing a Signer Class

Implementing the `Signer` class that we require follows virtually the same process:

```
package javasec.samples.appc;

import java.security.*;

public class XYZSigner extends Signer {
    private int trustLevel;

    public XYZSigner(String name, IdentityScope scope)
                                throws KeyManagementException {
        super(name, scope);
        scope.addIdentity(this);
    }

    public void setPublicKey(PublicKey key)
                                throws KeyManagementException {
        IdentityScope scope = getScope(  );
        if (scope != null) {
            Identity i = getScope(  ).getIdentity(key);
            if (i != null && !equals(i))
                throw new KeyManagementException(
                                "Duplicate public key");
        }
        super.setPublicKey(key);
    }
    public void addCertificate(Certificate cert)
                                    throws KeyManagementException {
        IdentityScope scope = getScope(  );
        if (scope != null) {
            Identity i = getScope().getIdentity(cert.getPublicKey(  ));
            if (i != null && !equals(i))
                throw new KeyManagementException(
                                    "Duplicate public key");
        }
        super.addCertificate(cert);
    }

    public int getTrust(  ) {
        return trustLevel;
    }

    void setTrust(int x) {
        if (x < 0 || x > 10)
            throw new IllegalArgumentException("Invalid trust level");
        trustLevel = x;
    }

    public String toString(  ) {
        return super.toString(  ) + " trust level: " + trustLevel;
    }
}
```

We do not need to provide an overridden method for the `setKeyPair( )` method of the `Signer` class to ensure that a duplicate private key is not inserted into the identity scope. Since we can only insert a private key with a public key, and since there is a one−to−one correspondence between such keys, we know that if the public keys are unique, the private keys are unique as well.

C.4.3 A Shared System Identity Scope

In the architecture we're examining, there are two identity scopes:

- The private scope. This scope will hold one and only one instance of `XYZSigner`. This signer represents the user who owns that particular database.
- The public scope. This scope will hold several instances of `XYZIdentity` but no signers −− since it is to be shared, we don't want it to contain any private keys.

Each of these scopes will be an instance of the `XYZFileScope` that we showed earlier. To combine them, we'll create another identity scope that holds a reference to both scopes:

```
package javasec.samples.appc;

import java.security.*;
import java.util.*;

public class XYZIdentityScope extends IdentityScope {
    private transient IdentityScope publicScope;
    private transient IdentityScope privateScope;

    public XYZIdentityScope(  ) throws KeyManagementException {
        super("XYZIdentityScope");
        privateScope = new XYZFileScope("/floppy/floppy0/private");
        publicScope = new XYZFileScope("/auto/shared/sharedScope");
        setSystemScope(this);
    }
    public int size(  ) {
        return publicScope.size() + privateScope.size(  );
    }

    public Identity getIdentity(String name) {
        Identity id;
        id = privateScope.getIdentity(name);
        if (id == null)
            id = publicScope.getIdentity(name);
        return id;
    }

    public Identity getIdentity(PublicKey key) {
        Identity id;
        id = privateScope.getIdentity(key);
        if (id == null)
            id = publicScope.getIdentity(key);
        return id;
    }

    public void addIdentity(Identity identity)
                                throws KeyManagementException {
        throw new KeyManagementException(
                    "This scope does not support adding identities");
    }

    public void removeIdentity(Identity identity)
                                throws KeyManagementException {
        throw new KeyManagementException(
                    "This scope does not support removing identities");
    }

    class XYZIdentityScopeEnumerator implements Enumeration {
        private boolean donePrivate = false;
        Enumeration pubEnum = null, privEnum = null;

        XYZIdentityScopeEnumerator(  ) {
            pubEnum = publicScope.identities(  );
            privEnum = privateScope.identities(  );
            if (!privEnum.hasMoreElements(  ))
                donePrivate = true;
        }

        public boolean hasMoreElements(  ) {
            return pubEnum.hasMoreElements(  ) ||
                   privEnum.hasMoreElements(  );
        }

        public Object nextElement(  ) {
            Object o = null;
            if (!donePrivate) {
                o = privEnum.nextElement(  );
                if (!privEnum.hasMoreElements(  ))
                    donePrivate = true;
            }
            else o = pubEnum.nextElement(  );
            if (o == null)
                throw new NoSuchElementException(
                            "XYZIdentityScopeEnumerator");
            return o;
        }
    }
    public Enumeration identities( ) {
        return new XYZIdentityScopeEnumerator( );
    }
}
```

The idea behind this class is that it is going to hold identities containing private keys and that those private keys should be held somewhere safe. For this example, we're assuming that the private identity scope database will be stored on a floppy disk somewhere −− that way, a user can move the identity scope around with her, and the private key won't be left on a disk where some malicious person might attempt to retrieve it.

This class is completely tailored to a Solaris machine since we've hardwired the name of the private file to a file on the default floppy drive of a Solaris machine, and we've hardwired the name of the public file to a file that can be automounted on the user's machine. On other machines, the name of the floppy drive will vary, and a complete implementation of this class would really require the filenames to be properties, which could be set to the appropriate values for the hardware on which the Java virtual machine is running. The public database probably shouldn't even be a file; it should be held on a remote machine somewhere and accessed via RMI or another technique. We'll leave those enhancements as an exercise for the reader.

Now that we have the two scopes we're interested in, completing the implementation is a simple matter of:

- Setting this identity scope to be the system identity scope. This allows the developer to use the standard methods we've already seen to extract information from this scope.
- Overriding the `getIdentity( )` and `identities( )` methods so that they operate on both included identity scopes. Remember that often identity scopes are disjoint; in this case, however, it makes sense for there to be a single interface to the two identity scopes.
- Overriding the `addIdentity( )` and `removeIdentity( )` methods to prevent them from changing the underlying identity databases. We'll see how to manipulate the individual database in the next section.

C.4.4 Creating Identities

The XYZ Corporation is concerned about two sorts of identities: identities from corporations and individuals outside the corporation and identities of employees. The latter must all have private keys in order for the employees to be able to sign documents and will be instances of the `XYZSigner` class; the former need only public keys and will be instances of the `XYZIdentity` class.

In order to create these identities, we're going to rely on the facilities provided by `javakey` to do the bulk of the work for us, then we're going to read the generic entity out of the javakey database and turn it into an XYZ−based entity. This allows us to import or create certificates for these identities, which is something that only `javakey` can do in Java 1.1.

When a new employee comes to the XYZ Corporation, we must generate a private identity database for that employee on a floppy that can be given to the employee. As a first step, however, we must create the employee in a standard javakey database so that the employee can be given a certificate to accompany her identity. Once we've got the employee into the javakey database, here's the code we use to convert the `javakey` entry into the `XYZIdentityScope` we just examined:

```
package javasec.samples.appc;

import java.security.*;

public class NewEmployee {
    public static void main(String args[]) {
        try {
            IdentityScope is = IdentityScope.getSystemScope(  );
            Signer origSigner = (Signer) is.getIdentity(args[0]);

            System.out.println(
                        "Please insert the floppy for " + args[0]);
            System.out.print("Press enter when ready:  ");
            System.in.read(  );
            XYZFileScope privateScope =
                        new XYZFileScope("/floppy/floppy0/private");
            XYZSigner newSigner = new XYZSigner(args[0], privateScope);
            KeyPair kp = new KeyPair(origSigner.getPublicKey(  ),
                                     origSigner.getPrivateKey(  ));
            newSigner.setKeyPair(kp);
            newSigner.setInfo(origSigner.getInfo(  ));
            Certificate certs[] = origSigner.certificates(  );
            for (int i = 0; i < certs.length; i++)
                newSigner.addCertificate(certs[i]);
            newSigner.setTrust(Integer.parseInt(args[1]));
            privateScope.save(  );

            XYZFileScope sharedScope =
                        new XYZFileScope("/auto/shared/sharedScope");
            XYZIdentity newId = new XYZIdentity(args[0], sharedScope);
            newId.setPublicKey(origSigner.getPublicKey(  ));
            newId.setInfo(origSigner.getInfo(  ));
            certs = origSigner.certificates(  );
            for (int i = 0; i < certs.length; i++)
                newId.addCertificate(certs[i]);
            newId.setTrust(Integer.parseInt(args[1]));
            sharedScope.save(  );
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

This program is then run with the name of the employee as an argument. When the program is run, two things happen:

1. The correct private key database is created and written to the floppy. The private key database has the signing identity of the new employee loaded into it.
2. The shared public database is opened, and the identity of the new employee is added to it.

In both cases, it was necessary to read the existing data out of the entity read from the javakey database and convert that data into an XYZ−based class. We could have used the existing object (a subclass of the `Identity` or `Signer` class), but that would not have allowed us to associate a level of trust with these entities in our database. After the program has run, both databases have the desired entity, with the desired set of keys.

When the system administrator for the XYZ Corporation receives a public key (and a certificate) for an entity that is not going to be a signer within the XYZ Corporation, a similar procedure would need to be followed to enter the certificate into the javakey database, and then extract out the new identity and update only the shared identity scope. Code to do that would be very similar to the code shown previously.

C.5 Summary

In this appendix, we have shown an example of an identity−based key management system. Such a system is the only choice for key management for developers in Java 1.1.

The identity−based key management system does have one advantage: it allows the retrieval of identity objects from the database while the keystore−based system only allows for retrieval of keys and certificates. This means that an identity−based system can embed within it other information about an entity (including, for example, a level of trust associated with that individual); this other information is available to users of the database in a straightforward way.

Appendix D. The Secure Java Container

In this appendix, we'll outline a container that is capable of running Java programs securely. The term container is most often used to refer to a Java 2 Enterprise Edition (J2EE) application server; the application server provides an environment in which you run your programs. But containers need not be J2EE application servers: Java−enabled browsers are applet containers, the Java command line sets up a J2SE application container, and so on. A container is just a shell that runs other code.

In Java 2, when you execute the Java program, you're actually starting a container that is referred to as the launcher. The launcher is set up to run applications security within the Java 2 framework (assuming that you've specified the `−Djava.security.manager` option). In order to achieve something similar in Java 1.1, you have to write a container from scratch; the container is responsible for setting a security manager, using an appropriate class loader, and so on.

Hence, the container that we outline in this appendix is most appropriate for Java 1.1. However, there are times even in Java 2 when you might want to write your own container for standard edition applications. If you want to use a different policy class, for instance, you must instantiate and register the policy class, instantiate a new class loader, and then execute your program from the new class loader. Similarly, you may want to install a new security manager that doesn't use (or supplements the use of) the access controller, particularly for things like thread permissions that don't follow a permission−based model very well. The container in this chapter can be adapted in Java 2 to do all of these things.

Note that if you're using a 1.1 VM that is embedded into a browser, this code won't help you; what we're showing is essentially the code that is built into the browser (except that our code will run applications, not applets). To change a browser security model in 1.1 requires browser−specific code.

From a security perspective, the linchpin of a 1.1−based application is the security manager, so we'll spend a lot of time discussing techniques for implementing that. And since the 1.1 security manager is closely tied to the class loader, we'll develop a 1.1 class loader. We'll tie all of this together into a program called `JavaRunner`, which you can use to run 1.1−based applications securely.

D.1 The 1.1−Based Class Loader

We'll take a bottom up approach in this example, so we'll start with the class loader. In 1.1, writing a class loader means extending the `ClassLoader` class and overriding its `loadClass( )` method to provide the necessary semantics.

The implementation that we'll use follows the same steps as we listed in Chapter 6. However, some of those steps rely on methods of the class loader that we haven't yet examined. In Java 2, these methods aren't used because they're called automatically by the `loadClass( )` method; in 1.1, you're responsible for calling the following methods:

protected final Class findSystemClass(String name)

Attempt to find the named class by using the internal class loader to search the user's classpath and the core JDK classes (*classes.zip*). If the system class is not found, a `ClassNotFoundException` is generated. This accomplishes a similar task as deferring to the parent class loader does in Java 2 (although the Java 2 mechanism is much more flexible).

protected final Class findLoadedClass(String name)

Find the class object for a class previously loaded by this class loader. This method returns `null` if it cannot find the given class.

protected final void resolveClass(Class c)

For a given class, resolve all the immediately needed class references for the class; this will result in recursively calling the class loader to ask it to load the referenced class.

In addition to implementing the `loadClass( )` method, we need our class loader to provide help to the security manager. There are three things the security manager will need to know:

- Whether or not a particular class should be trusted. There are many ways to implement this, but one way is to let the class loader make that decision. Our class loader will make that decision based on the location from which the class was loaded; it could instead make that decision based on the signers of the class or any other information.
- The location from which a class was loaded. That's needed because the security manager will always allow classes to make a connection back to the host from which they were loaded.
- Which thread group a new thread should belong to. The default thread group will be based on the class loader instance so that threads can be partitioned based on the application that created them.

Finding Previously Loaded Classes

According to the Java specification, a class loader is required to cache the classes that it has previously loaded so that when it is asked to load a particular class, it is not supposed to reread the class file. Not only is this more efficient, but it also allows a simpler internal implementation of many methods, including the `resolveClass( )` method.

The Java specification hedges this somewhat by stating that this requirement may change in the future, when the classes will be cached by the virtual machine itself. Hence, the `ClassLoader` class in Java 1.0 did not do any caching, and it was up to concrete implementations of class loaders to perform this caching.

Beginning with Java 1.1, however, caching within the class loader was considered important enough that the base `ClassLoader` class now performs this caching automatically: a class is put into the cache of the class loader in the `defineClass( )` method and may be retrieved from the cache with the `findLoadedClass( )` method. Since these methods are final, and since the cache itself is a private instance variable of the `ClassLoader` class, this permits a class loader to be written without any knowledge of whether the class loader or the virtual machine is doing the caching.

The class loader is the only class suited to make these decisions since the class loader can easily keep track of information like where the class came from.

Here's the complete implementation of our 1.1−based class loader:

```
package javasec.samples.appd;

import java.io.*;
import java.net.*;
import java.util.*;

public class JRClassLoader extends ClassLoader {

    private URL urlBase;
    private static URL trustedBase;
    private Hashtable trustedClasses;
    private static int groupNum;
    private ThreadGroup threadGroup;

    static {
        String s = System.getProperty("trustedBase");
        if (s != null)
            try {
                trustedBase = new URL(s);
            } catch (Exception e) {
                throw new IllegalArgumentException(
                                 "Bad value for trustedBase " + s);
            }
    }

    public JRClassLoader(String base) {
        try {
            if (!(base.endsWith("/")))
                base = base + "/";
        urlBase = new URL(base);
        } catch (Exception e) {
            throw new IllegalArgumentException(base);
        }
        trustedClasses = new Hashtable(13);
    }

    protected Class loadClass(String name, boolean resolve) {
        Class c;
        SecurityManager sm = System.getSecurityManager(  );

        // Step 1 -- Check to make sure that we can access this class
        if (sm != null) {
            int i = name.lastIndexOf('.');
            if (i >= 0)
                sm.checkPackageAccess(name.substring(0, i));
        }

         // Step 2 -- Check for a previously loaded class
         c = findLoadedClass(name);
         if (c != null)
            return c;

        // Step 3 -- Check for system class first
        try {
            c = findSystemClass(name);
            return c;
        } catch (ClassNotFoundException cnfe) {
            // Not a system class, simply continue
        }

        // Step 4 -- Check to make sure that we can define this class
        if (sm != null) {
            int i = name.lastIndexOf('.');
            if (i >= 0)
                sm.checkPackageDefinition(name.substring(0, i));
        }

        // Step 5 -- Read in the class file
        byte data[] = lookupData(name);
        if (data == null)
             return null;
        // Steps 6 and 7 -- Define the class from the data; this also
        //         passes the data through the byte code verifier
        c = defineClass(name, data, 0, data.length);
        if (trustedClasses.get(name) != null) {
            trustedClasses.remove(name);
            trustedClasses.put(c, name);
        }

        // Step 8 -- Resolve the internal references of the class
        if (resolve)
            resolveClass(c);

        return c;
    }

    byte[] lookupData(String n) {
        try {
            byte[] b = null;
            String urlName = n.replace('.', '/');

            if (trustedBase != null) {
                URL url = new URL(trustedBase, urlName + ".class");
                b = readData(url);
            }

            if (b == null) {
                URL url = new URL(urlBase, urlName + ".class");
                b = readData(url);
            }
            else trustedClasses.put(n, "dummy");

            return b;
        } catch (Exception e) {
            return null;
        }
    }

    byte[] readData(URL url) {
        try {
            InputStream is = url.openConnection().getInputStream(  );
            BufferedInputStream bis = new BufferedInputStream(is);
            ByteArrayOutputStream baos = new ByteArrayOutputStream(  );
            boolean eof = false;
            while (!eof) {
                int i = bis.read(  );
                if (i == -1)
                    eof = true;
                else baos.write(i);
            }
            return baos.toByteArray(  );
        } catch (Exception e) {
            return null;
        }
    }

    boolean getTrusted(Class c) {
        return trustedClasses.get(c) != null;
    }

    private static synchronized int getGroupNum(  ) {
        return groupNum++;
    }
    synchronized ThreadGroup getThreadGroup(  ) {
        if (threadGroup == null)
            threadGroup = new ThreadGroup("JavaRunner ThreadGroup-"
                                          + getGroupNum(  ));
        return threadGroup;
    }

    String getHost(  ) {
        return urlBase.getHost(  );
    }
}
```

This class loader is set up to load classes from two locations. One is specified by the URL that is passed to its constructor; the other is specified by the system property `trustedBase`. When the `loadClass( )` method is called, it performs the required bookkeeping before calling the `lookupData( )` method; this method first tries to load the class from the trusted base and then -- if that fails -- from the URL that was passed to its constructor.

Notice that when the `lookupData( )` succeeds in finding the class when reading from the trusted base that it saves the class name in the `trustedClasses` hashtable. Later, the `loadClass( )` replaces that mapping with the actual class object so that the `getTrusted( )` method can perform a simple lookup on that hashtable. The point of the `getTrusted( )` method is that the security manager will allow any classes marked as trusted -- in this case, classes loaded from the `trustedBase` URL -- to perform any actions.

D.1.1 Defining Signed Classes

Handling signed classes in 1.1 is hard because the classes that read a 1.1 signed jar file are not public. The easiest way to get around that is to use a different tool to sign the classes. You can put the classes into a guarded object (such as we did in Chapter 12) and have the certificate accompany the classes that way. If you're ambitious, you can decode the PKCS #7 signature block in the signed jar file.

However you decide to do it, you must create an array of identity objects; each object in the array must contain the public key of the entity that signed the particular class. You then use these two methods:

protected final void setSigners(Class c, Object[] signers)

Associate the list of signers with the given class. Although it's defined as an arbitrary array of objects, each element of the signers array must be an `Identity` object, or other parts of the Java API will not work. This is a method of the `ClassLoader` class and must be called by your class loader.

public native Object[] getSigners( )

Return the list of signers associated with a class. This is a method of the `Class` class; you invoke it on a `Class` object to find out which entities signed that class.

If a class loader holds the appropriate array of identities in the variable `ids`, then this is the code you'd need to use:

```
if (isSecure(urlName)) {
    cl = defineClass(name, buf, 0, buf.length);
    if (ids != null)
        setSigners(cl, ids);
}
else cl = defineClass(name, buf, 0, buf.length);
```

The `isSecure( )` method in this case must base its decision on information obtained from reading the signed class and verifying the signature accompanies it. The array of `ids` will need to be created by constructing instances of the `Identity` class to represent the signer of the class.

The reason for setting the signers in this way is to allow the security manager to retrieve those signatures easily. When the security manager does not defer all permissions to the access controller −− and, hence, in all Java 1.1 programs −− the security manager will need to take advantage of signed class information as a basis for its decisions. This is typically done by programming the security manager to retrieve the keys that were used to sign a class via the `getSigners( )` method. This allows the security manger to function with any standard signature−aware class loader. The security manager could then do something like this:

```
public void checkAccess(Thread t) {
    Class cl = currentLoadedClass(  );
    if (cl == null)
        return;
    Identity ids[] = (Identity[]) cl.getSigners(  );
    for (int i = 0; i < ids.length; i++) {
        if (isTrustedId(ids[i]))
            return;
    }
    throw new SecurityException("Can't modify thread states");
}
```

The key to this example is writing a good `isTrustedId( )` method. A possible implementation is to use the information stored in the identity database to grant a level of trust to an entity; such an implementation requires that you have a non−default implementation of these databases (such as we showed in Appendix C). Alternately, your application could hardwire the public keys of certain entities (like the public key of the HR group of XYZ corporation) and use that information as the basis for its security decisions.

D.2 The 1.1−Based Security Manager

In order to write a complete 1.1−based security manager, we must extend the `SecurityManager` class and use a number of its protected methods to determine certain information. In 1.1, the default implementation of each public method of the `SecurityManager` class is to throw an exception, so we must provide an implementation of all the methods we listed in Chapter 4. The security manager that we write will have a binary notion of trusted and untrusted classes.

D.2.1 Protected Methods of the Security Manager

The distinction that our security manager makes between trusted and untrusted code has its roots in information that the security manager must obtain from the class loader. We've seen part of one way that happens: the class loader can provide an agreed−upon interface that the security manager uses to obtain certain information. The second way that happens is by using the protected methods of the security manager; they are summarized in Table D−1. Use of these methods is discouraged in Java 2; most of them are officially deprecated, and the remainder should be avoided.

Table D−1. Protected Methods of the Security Manager Class

Method	Purpose
`getClassContext(  )`	

Return all the classes on the stack to see who has called us

`currentClassLoader( )`

Return the most recent class loader

`currentLoadedClass( )`

Return the class that was most recently loaded with a class loader

`classLoaderDepth( )`

Return the depth in the call stack where the most recent class loader was found

`classDepth( )`

Return the depth in the call stack of the given class

`inClass( )`

Return `true` if the given class is on the stack

`inClassLoader( )`

Return `true` if any class on the stack came from a class loader

We'll discuss each of these methods, starting with the `getClassContext( )` method. This method itself is rarely used in a security manager, but it is the basis for many of the methods we'll discuss in this section.

protected native Class[] getClassContext( )

Returns an array of `Class` objects in the order of the call stack for the current method. The first element of the array is always the `Class` object for the security manager class, the second element is the `Class` object for the method that called the security manager, and so on.

Accessing all the classes in this array is one way to determine whether the call originally came from code that is in the Java API or whether it came from other code. For example, we could put the following method into our custom security manager:

```
package javasec.samples.appd;

public class MySecurityManager extends SecurityManager {
    public void checkRead(String s) {
        Class c[] = getClassContext(  );
        for (int i = 0; i < c.length; i++) {
            String name = c[i].getName(  );
            System.out.println(name);
        }
    }
}
```

If we then try to create a `FileReader` object:

```
package javasec.samples.appd;

import java.io.*;

public class TestSecurityManager {
    public static void main(String args[]) throws Exception {
        System.setSecurityManager(new MySecurityManager(  ));
        FileReader f = new FileReader("TestSecurityManager.java");
    }
}
```

we see the following output from the `checkRead( )` method:

```
javasec.samples.appd.MySecurityManager
java.io.FileInputStream
java.io.FileReader
javasec.samples.appd.TestSecurityManager
```

In other words, a method in the `Test` class invoked a method in the `FileReader` class, which invoked a method in the `FileInputStream` class, which invoked a method (the `checkRead( )` method, in fact) in the `MySecurityManager` class.

The policies you want to enforce determine how you use the information about these classes −− just keep in mind that the first class you see is always your security manager class and the second class you see is normally some class of the Java API. This last case is not an absolute −− it's perfectly legal, though rare, for any arbitrary class to call the security manager. And as we saw in Chapter 4, some methods are called by platform−specific classes that implement particular interfaces of the Java API (such as methods that implement the `Toolkit` class).

Also keep in mind that there may be several classes from the Java API returned in the class array −− for example, when you construct a new thread, the `Thread` class calls the `checkAccess( )` method; the classes returned from the `getClassContext( )` method in that case are:

```
javasec.samples.appd.MySecurityManager
java.lang.Thread
java.lang.Thread
java.lang.Thread
java.lang.Thread
javasec.samples.appd.TestSecurityManager
javasec.samples.appd.TestSecurityManager
```

We get this output because the `Thread` class constructor calls three other internal methods before it calls the security manager. Our `TestSecurityManager` class has created a thread in an internal method as well, so that class also appears twice in the class array.

protected native ClassLoader currentClassLoader( )

Search the array of classes returned from the `getClassContext( )` method for the most recently called class that was loaded via a program−defined class loader, and return that class loader.

The objects in the class array returned from the `getClassContext( )` method are generally used to inspect the class loader for each class −− that's how the security manager can make a policy decision about classes that were loaded from disk versus classes that were loaded from the network (or elsewhere). The simplest test that we can make is to see if any of the classes involved in the current method invocation are loaded from the network, in which case we can deny the attempted operation. This is the method we use to do that.

To understand `currentClassLoader( )`, we need to understand how the 1.1 class loader works. The class loader first calls the `findSystemClass( )` method, which attempts to find the class in the user's classpath. If that call is unsuccessful, the class loader loads the class in a different manner (e.g., by loading the class over the network). As far as the Java virtual machine is concerned, the class loader associated with a class that was loaded via the `findSystemClass( )` method is `null`. If an instance of the `ClassLoader` class defined the class (by calling the `defineClass( )` method), then (and only then) does Java make an association between the class and the class loader. This association is made by storing a reference to the class loader within the class object itself; the `getClassLoader( )` method of the `Class` object can be used to retrieve that reference.

Hence, the `currentClassLoader( )` method is equivalent to:[A]

> [A] The truth is that the `currentClassLoader( )` method is written in native code, so we don't know how it actually is implemented, but it is functionally equivalent to the code shown. This is true about most of the methods of this section, which for efficiency reasons are written in native code.

```
protected ClassLoader currentClassLoader(  ) {
    Class c[] = getClassContext(  );
    for (int i = 1; i < c.length; i++)
        if (c[i].getClassLoader(  ) != null)
            return c[i].getClassLoader(  );
    return null;
}
```

We can use this method to disallow writing to a file by any class that was loaded via a class loader:

```
public void checkWrite(String s) {
    if (currentClassLoader(  ) != null)
        throw new SecurityException("checkWrite");
    }
}
```

With this version of `checkWrite( )`, only the Java virtual machine can open a file for writing. When the Java virtual machine initializes, for example, it may create a thread for playing audio files. This thread will attempt to open the audio device on the machine by instantiating one of the standard Java API file classes. When the instance of this class is created, it (as expected) calls the `checkWrite( )` method, but there is no class loader on the stack. The only methods that are involved in the thread opening the audio device are methods that were loaded by the Java virtual machine itself and hence have no class loader. Later, however, if an applet class tries to open up a file on the user's machine, the `checkWrite( )` method is called again, and this time there is a class loader on the stack: the class loader that was used to load the applet making the call to open the file. This second case will generate the security exception.

A number of convenience methods of the security manager class also relate to the current class loader:

protected boolean inClassLoader( )

Test to see if there is a class loader on the stack:

```
protected boolean inClassLoader(  ) {
        return currentClassLoader(  ) != null;
    }
```

protected Class wcurrentLoadedClass( )

Return the class on the stack that is associated with the current class loader:

```
protected Class currentLoadedClass(  ) {
    Class c[] = getClassContext(  );
    for (int i = 0; i < c.length; i++)
        if (c[i].getClassLoader(  ) != null)
            return c[i];
    return null;
}
```

protected native int classDepth(String name)

Return the index of the class array from the `getClassContext( )` method where the named class is found:

```
protected int classDepth(String name) {
    Class c[] = getClassContext(  );
    for (int i = 0; i < c.length; i++)
        if (c[i].getName(  ).equals(name))
            return i;
    return −1;
}
```

protected boolean inClass(String name)

Indicate whether the named class is anywhere on the stack:

```
protected boolean inClass(String name) {
    return classDepth(name) >= 0;
}
```

Many of these convenience methods revolve around the idea that an untrusted class may have called a method of a trusted class and that the trusted class should not be allowed to perform an operation that the untrusted class could not have performed directly. These methods allow you to write a Java application made up of trusted classes that itself downloads and runs untrusted classes. The `appletviewer` is the best−known example of this sort of program. For example, the security manager of the `appletviewer` does not allow an arbitrary applet to initiate a print job, but the `appletviewer` itself can.

The `appletviewer` initiates a print job when the user selects the "Print" item from one of the standard menus. Since the request comes from a class belonging to the `appletviewer` itself (that is, the callback method of the menu item), it is initiating the request (at least as far as the security manager is concerned). An applet initiates the request when it tries to create a print job directly.

In both cases, the `getPrintJob( )` method of the `Toolkit` class calls the `checkPrintJobAccess( )` method of the security manager. The security manager must then look at the classes on the stack and determine if the operation should succeed. If there is an untrusted (applet) class anywhere on the stack, the print request started with that class and should be rejected; otherwise, the print request originated from the `appletviewer` classes and is allowed to proceed.

Note the similarity between this technique and the manner in which the access controller works, where the `appletviewer` classes would be in the system domain and the applet classes in a different domain.

D.2.1.1 The class loader depth

The example that we just gave is typical of the majority of security checks the security manager makes. You can often make a decision about whether or not an operation should be allowed simply by knowing whether or not there is a class loader on the stack since the presence of a class loader means that an untrusted class has initiated the operation in question.

There's a group of tricky exceptions to this rule, however, and those exceptions mean that you sometimes have to know the exact depth at which the class loader was found. Before we dive into those exceptions, we must emphasize that the use of the class loader depth is not pretty, and it is very dependent on each release of Java. The rules that we're about to give will not necessarily apply to any release of Java except 1.1.

The depth at which the class loader was found in the class context array can be determined by this method:

protected native int classLoaderDepth( )

Return the index of the class array from the `getClassContext( )` method where the current class loader is found:

```
protected int classLoaderDepth(  ) {
    Class c[] = getClassContext(  );
        for (int i = 0; i < c.length; i++) {
            if (c[i].getClassLoader(  ) != null)
                return i;
        }
    return -1;
}
```

Let's look at this method in the context of the following applet:

```
package javasec.samples.appd;

import java.applet.*;
import java.math.*;

public class DepthTest extends Applet {
    native void evilInterface(  );

    public void init(  ) {
        doMath(  );
        infiltrate(  );
    }

    public void infiltrate(  ) {
        try {
            System.loadLibrary("evilLibrary");
            evilInterface(  );
        } catch (Exception e) {}
    }

    public void doMath(  ) {
        BigInteger bi = new BigInteger("100");
        bi = bi.add(new BigInteger("100"));
        System.out.println("answer is " + bi);
    }
}
```

Under normal circumstances, we would expect the `doMath( )` method to inform us (rather inefficiently) that 100 plus 100 is 200. We would further expect the call to the `infiltrate( )` method to generate a security exception since an untrusted class is not normally allowed to link in a native library.

The security exception in this case is generated by the `checkLink( )` method of the security manager. When the `infiltrate( )` method calls the `System.loadLibrary( )` method, the `loadLibrary( )` method in turn calls the `checkLink( )` method. If we were to retrieve the array of classes (via the `getClassContext( )` method) that led to the call to the `checkLink( )` method, we'd see the following classes on the stack:

```
javasec.samples.appd.MySecurityManager (the checkLink(  ) method)
java.lang.Runtime                      (the loadLibrary(  ) method)
java.lang.System                       (the loadLibrary(  ) method)
javasec.samples.appd.DepthTest         (the infiltrate(  ) method)
javasec.samples.appd.DepthTest         (the init(  ) method)
... other classes from the browser ...
```

Because the untrusted class `DepthTest` appears on the stack, we are tempted to reject the operation and throw a security exception.

Life is not quite that simple in this case. As it turns out, the `BigInteger` class contains its own native methods and hence depends on a platform−specific library to perform many of its operations. When the `BigInteger` class is loaded, its static initializer attempts to load the math library (by calling the `System.loadLibrary( )` method), which is the library that contains the code to perform these native methods.

Because of the way in which Java loads classes, the `BigInteger` class is not loaded until it is actually needed −− that is, until the `doMath( )` method of the `DepthTest` class is called. If you recall our discussion regarding how the class loader works, you'll remember that when the `doMath( )` method is called and needs access to the `BigInteger` class, the class loader that created the `DepthTest` class is asked to find that class (even though the `BigInteger` class is part of the Java API itself). Hence, the applet class loader (that is, the class loader that loaded the `DepthTest` class) is used to find the `BigInteger` class, which it does by calling the `findSystemClass( )` method. When the `findSystemClass( )` method loads the `BigInteger` class from disk, it runs the static initializers for that class, which call the `System.loadLibrary( )` method to load in the math library.

The upshot of all this is that the `System.loadLibrary( )` method calls the security manager to see if the program in question is allowed to link in the math library. This time, when the `checkLink( )` method is called, the class array from the `getClassContext( )` method looks like this:

```
javasec.samples.appd.MySecurityManager (the checkLink(  ) method)
java.lang.Runtime                      (the loadLibrary(  ) method)
java.lang.System                       (the loadLibrary(  ) method)
java.math.BigInteger                   (the static intializer)
java.lang.ClassLoader                  (the findSystemClass(  ) method)
AppletLoader                           (the loadClass(  ) method)
java.lang.ClassLoader            (the loadClassInternal(  ) method)
javasec.samples.appd.DepthTest   (the doMath(  ) method)
javasec.samples.appd.DepthTest   (the init(  ) method)
... various browser classes ...
```

As we would expect, the first three elements of this list are the same as the first three elements of the previous list −− but after that, we see a radical difference in the list of classes on the stack. In both cases, the untrusted class (`DepthTest`) is on the stack, but in this second case, it is much further down the stack than it was in the first case. In this second case, the untrusted class indirectly caused the native library to be loaded; in the first case the untrusted class directly requested the native library to be loaded. That distinction is what drives the use of the `classLoaderDepth( )` method.

So in this example, we need the `checkLink( )` method to obtain the depth of the class loader (that is, the depth of the first untrusted class on the stack) and behave appropriately. If that depth is 3, the `checkLink( )` method should throw an exception, but if that depth is 7, the `checkLink( )` method should not throw an exception. There is nothing magical about a class depth of 7, however −− that just happens to be the depth returned by the `classLoaderDepth( )` method in our second example. A different example might well have produced a different number, depending on the classes involved.

There is, however, something special about a class depth of 3 in this example: a class depth of 3 always means that the untrusted class called the `System.loadLibrary( )` method, which called the `Runtime.loadLibrary( )` method, which called the security manager's `checkLink( )` method.[B] Hence, when there is a class depth of 3, it means that the untrusted class has directly attempted to load the library. When the class depth is greater than 3, the untrusted class has indirectly caused the library to be loaded. When the class depth is 2, the untrusted class has directly called the `Runtime.loadLibrary( )` method −− which is to say again that the untrusted class has directly attempted to load the library. When there is a class depth of 1, the untrusted class has directly called the `checkLink( )` method −− which is possible, but that is a meaningless operation. So in this case, a class depth that is 3 or less (but greater than −1, which means that no untrusted class is on the stack) indicates that the call came directly from an untrusted class and should be handled appropriately (usually meaning that a security exception should be thrown).

> [B] Theoretically, it could also mean that an untrusted class has called a trusted class that has called the `Runtime.loadLibrary( )` method directly. However, the Java API never bypasses the `System.loadLibrary( )` method, so that will not happen in practice. If you expect trusted classes in your Java application to work under the scenario we're discussing here, you must also follow that rule.

But while 3 is a magic number for the `checkLink( )` method, it is not necessarily a magic number for all other methods. In general, for most methods the magic number that indicates that an untrusted class directly attempted an operation is 2: the untrusted class calls the Java API, which calls the security manager. Other classes have other constraints on them that change what their target number should be.

The class depth is therefore a tricky thing: there is no general rule about the allowable class depth for an untrusted class. Worse, there's no assurance that the allowable class depth may not change between releases of the JDK −− the JDK could conceivably change its internal algorithm for a particular operation to add another method call, which would increase the allowable class depth by 1. This is one reason why the class depth is such a bad idea: it requires an intimate knowledge of all the trusted classes in the API in order to pick an appropriate class depth. Worse, a developer may introduce a new method into a call stack and completely change the class depth for a sensitive operation without realizing the effect this will have on the security manager.

Nonetheless, in order for certain classes of the Java API to work correctly, you need to put the correct information into your 1.1−based security manager (such as in the `checkLink( )` method that we just examined). The methods that need such treatment are summarized in Table D−2.

Table D−2. Methods of the SecurityManager Class Affected by the Depth of the Class Loader

Method	Depth to Avoid

Remarks

`checkCreateClassLoader( )`

2

Java beans create a `SystemClassLoader`

`checkPropertiesAccess( )`

2

Java API calls `System.getProperties( )`

`checkPropertyAccess( )`

2

Java API gets many properties

`checkAccess(Thread t)`

3

Java API manipulates its own threads

`checkAccess(ThreadGroup tg)`

3

(sometimes 4)

Java API manipulates its own thread groups

`checkLink( )`

2 or 3

Java API loads many libraries

`checkMemberAccess( )`

3

Java API uses method reflection

`checkExec( )`

	2	Toolkit implementations of `getPrintJob( )` may execute a print command
`checkExit( )`	2	The application may call `exit`
`checkWrite( )`	2	Toolkit implementations may create temporary files; the Java API needs to write to audio and other device files
`checkDelete( )`	2	Toolkit implementations may need to delete temporary files
`checkRead( )`	2	Java API needs to read property files
`checkTopLevelWindow( )`	3	Trusted classes may need pop−up windows

In all cases in Table D−2, the Java API depends on being allowed to perform an operation that would normally be rejected if an untrusted class performed it directly. The JavaBeans classes, for example, create a class loader (an instance of `SystemClassLoader`) in order to abstract the primordial class loader. So if an untrusted class creates a Java bean, that Java bean must in turn be allowed to create a class loader, or the bean itself won't work.

Note that not every target depth in this table is 2. In the case of the `Thread` and `ThreadGroup` classes, operations that affect the state of the thread call the `checkAccess( )` method of the `Thread` or `ThreadGroup` class itself, which in turn calls the `checkAccess( )` method of the `SecurityManager` class. This extra method call results in an extra method on the stack and effectively increases the target depth by 1. Similarly, the `checkTopLevelWindow( )` method is called from the constructor of the `Window` class, which in turn is called from the constructor of the `Frame` class, resulting in a target depth of 3.

Remember that this table only summarizes the methods of the security manager where the actual depth of the class loader matters to the core Java API. If you're writing your own application, you need to consider whether or not your application classes want to perform certain operations. If you want classes in your application to be able to initiate a print job, for example, and you don't want untrusted classes that your application loads to initiate a print job, you'll want to put a depth check of 2 into the `checkPrintJobAccess( )` method. In general, for methods that aren't listed in the above table, a depth of 2 is appropriate if you want your application classes (i.e., classes from the classpath) to be able to perform those operations.

There is once again a nice similarity between these ideas and the access controller. When you call the `doPrivileged( )` method of the access controller, you're achieving the same thing a security manager achieves by testing the class depth. The point to remember about the class depth is that it allows the security manager to grant more permissions to a class than it would normally have -- just like the `doPrivileged( )` method grants its permissions to all protection domains that have previously been pushed onto the stack.

D.2.1.2 Protected instance variables in the security manager

There is a single protected instance variable in the security manager class, and that is the `inCheck` instance variable:

protected boolean inCheck

Indicate whether a check by the security manager is in progress.

The value of this variable can be obtained from the following method:

public boolean getInCheck( )

Return the value of the `inCheck` instance variable.

Since there is no corresponding public method to set this variable, it is up to the security manager itself to set `inCheck` appropriately.[C]

[C] Don't get all excited and think that your untrusted class can use this method to see when the security manager is working. As we'll see, it's only set by the security manager in a rare case, and even if it were set consistently, there's no practical way for your untrusted class to examine the variable during the short period of time it is set.

This variable has a single use: it must be set by the security manager before the security manager calls most methods of the `InetAddress` class. The reason for this is to prevent an infinite recursion between the security manager and the `InetAddress` class. This recursion is possible under the following circumstances.

1. An untrusted class attempts to open a socket to a particular host (e.g., *sun.com*). The expectation is that if the untrusted class was loaded from *sun.com* the operation will succeed; otherwise, the operation will fail.
2. Opening the socket results in a call to the `checkConnect( )` method, which must determine if the two hosts in question are the same. In the case of a class loaded from *sun.com* that is attempting to connect to *sun.com*, a simple string comparison is sufficient. If the names are the same, the `checkConnect( )` method can simply return immediately. In fact, this is the only logic performed by some browsers -- if the names do not literally match, the operation is denied immediately.
3. A complication arises if the two names do not match directly but may still be the same host. My machine has a fully qualified name of *piccolo.East.Sun.COM*; browsers on my local area network can access my machine's web server as *piccolo*, *piccolo.East*, or *piccolo.East.Sun.COM*. If the untrusted class is loaded from a URL that contained only the string *piccolo*, and the class attempts to open a

socket to *piccolo.East*, we may want that operation to succeed even though the names of the hosts are not equal.

Hence, the `checkConnect( )` method must retrieve the IP address for both names and compare those IP addresses.

4. To retrieve the IP address for a particular host, the `checkConnect( )` method must call the `InetAddress.getByName( )` method, which converts a string to an IP address.
5. The `getByName( )` method will not blithely convert a hostname to an IP address −− it will only do so if the program in question is normally allowed to make a socket connection to that host. Otherwise, an untrusted class could be downloaded into your corporate network and determine all the IP addresses that are available on the network behind your firewall. So the `getByName( )` method needs to call the `checkConnect( )` method in order to ensure that the program is allowed to retrieve the information that is being requested.

We see the problem here: the `getByName( )` method keeps calling the `checkConnect( )` method, which in turn keeps calling the `getByName( )` method. In order to prevent this recursion, the `checkConnect( )` method is responsible for setting the `inCheck` instance variable to `true` before it starts and then setting it to `false` when it is finished. Similarly, the `getByName( )` method is responsible for examining this variable (via the return from the `getInCheck( )` method); it does not call the `checkConnect( )` method if a security check is already in progress.

There may be other variations in this cooperation between the security manager and the `InetAddress` class −− other methods of the `InetAddress` class also use the information from the `getInCheck( )` method to determine whether or not to call the `checkConnect( )` method. But this is the only class where this information is used directly. You can set the `inCheck` method within other methods of your security manager, but there is no point in doing so.

If you implement a `checkConnect( )` method that calls the `InetAddress` class and sets the `inCheck` variable, you must make the `checkConnect( )` method and the `getInCheck( )` methods synchronized. This prevents another thread from directly looking up an IP address at the same time that the security manager has told the `InetAddress` class not to call the `checkConnect( )` method.

D.2.2 Implementation Techniques

We'll now turn our attention to implementing security policies. We'll list the complete security manager first and then explain the rationale behind some of its methods.

Here's the code:

```
package javasec.samples.appd;

import java.security.*;
import java.io.*;
import java.net.*;
import java.lang.reflect.*;
import java.util.*;

public class JavaRunnerManager extends SecurityManager {

    private void checkTrustedDepth(String err, int depth) {
        int clDepth = classLoaderDepth(  );
        if (clDepth == 0 || clDepth > depth + 1)
            return;

        ClassLoader cl = currentClassLoader(  );
        if (cl != null) {
            JRClassLoader jcl;
            try {
                jcl = (JRClassLoader) cl;
            } catch (ClassCastException cce) {
                // This can't happen unless our own classes
                // are out of sync
                throw new SecurityException("Unknown class loader");
            }
            if (!jcl.getTrusted(currentLoadedClass(  )))
                throw new SecurityException(err);
        }
    }

    public synchronized boolean getInCheck(  ) {
        return super.getInCheck(  );
    }

    public synchronized void checkConnect(String host, int port) {
        ClassLoader loader = currentClassLoader(  );
        String remoteHost;

        if (loader == null)
            return;
        if (!(loader instanceof JRClassLoader)) {
            System.err.println("Class loader out of sync");
            return;
        }

        JRClassLoader cl = (JRClassLoader) loader;
        remoteHost = cl.getHost(  );

        if (host.equals(remoteHost))
            return;

        try {
            inCheck = true;
            InetAddress hostAddr = InetAddress.getByName(host);
            InetAddress remoteAddr = InetAddress.getByName(remoteHost);
            inCheck = false;
            if (hostAddr.equals(remoteAddr))
                return;
        } catch (UnknownHostException uhe) {
            inCheck = false;
        }
        throw new SecurityException(
                  "Can't connect from " + remoteHost + " to " + host);
    }

    public void checkConnect(String host, int port, Object context) {
        checkConnect(host, port);
    }

    public void checkListen(int port) {
        if (inClassLoader(  ) && port < 1024 & port > −1)
            throw new SecurityException(
                         "Can't listen on privileged port");
    }

    public void checkAccept(String host, int port) {
        checkListen(port);
    }
```

```
public void checkMulticast(InetAddress maddr) { }
public void checkMulticast(InetAddress maddr, byte ttl) { }

public void checkPackageAccess(String pkg) { }
public void checkPackageDefinition(String pkg) {
    if (!pkg.startsWith("java.") && !pkg.startsWith("sun."))
        return;
    if (inClassLoader(  ))
        throw new SecurityException(
                          "Can't define sun/java classes");
}

public void checkAccess(Thread t) {
    ThreadGroup current = Thread.currentThread().getThreadGroup(  );
    if (!current.parentOf(t.getThreadGroup(  )))
        throw new SecurityException(
                          "Can't modify outside of group");
}

public void checkAccess(ThreadGroup tg) {
    ThreadGroup current = Thread.currentThread().getThreadGroup(  );
    if (!current.parentOf(tg))
        throw new SecurityException(
                          "Can't modify outside of group");
}

public void checkRead(String s) {
    checkTrustedDepth("checkread", 2);
}

public void checkRead(FileDescriptor fd) {
    if (!inClassLoader(  ))
        return;
    if (!fd.valid(  ) || !inClass("java.net.SocketInputStream"))
        throw new SecurityException("checkRead fd");
}

public void checkRead(String file, Object context) {
    checkRead(file);
}

public void checkWrite(FileDescriptor fd) {
    if (!inClassLoader(  ))
        return;
    if (!fd.valid(  ) || !inClass("java.net.SocketOutputStream"))
        throw new SecurityException("checkWrite fd");
}

public void checkWrite(String s) {
    checkTrustedDepth("checkwrite", 2);
}

public void checkDelete(String file) {
    checkRead(file);
}

public void checkCreateClassLoader(  ) {
    if (inClassLoader(  ))
        throw new SecurityException("createClassLoader");
}

public void checkExec(String cmd) {
    checkTrustedDepth("checkExec", 2);
    }

    public void checkExit(int status) {
        checkTrustedDepth("checkExit", 2);
    }

    public void checkLink(String lib) {
        checkTrustedDepth("checkExit", 3);
    }
    public void checkPropertiesAccess(  ) { }
    public void checkPropertyAccess(String key) { }
    public void checkPropertyAccess(String key, String def) { }

    public boolean checkTopLevelWindow(Object window) {
        try {
            checkTrustedDepth("top", 3);
        } catch (SecurityException se) {
            return false;
        }
        return true;
    }

    public void checkPrintJobAccess(  ) {
        checkTrustedDepth("printjob", 2);
    }

    public void checkSystemClipboardAccess(  ) {
        checkTrustedDepth("clipboard", 2);
    }

    public void checkAwtEventQueueAccess(  ) {
        checkTrustedDepth("eventqueue", 2);
    }

    public void checkSetFactory(  ) {
        checkTrustedDepth("setfactory", 2);
    }

    public void checkMemberAccess(Class clazz, int which) {
        checkTrustedDepth("member access", 2);
    }

    public void checkSecurityAccess(String provider) {
        checkTrustedDepth("security access", 2);
    }

    public ThreadGroup getThreadGroup(  ) {
        ClassLoader cl = currentClassLoader(  );
        if (cl == null || !(cl instanceof JRClassLoader))
            return super.getThreadGroup(  );
        JRClassLoader jcl = (JRClassLoader) cl;
        return jcl.getThreadGroup(  );
    }
}
```

There are a number of tests we want our security manager to reject if they are attempted directly by an untrusted class but which should succeed if they are attempted indirectly by an untrusted class. For these tests, we have to rely on the class depth to tell us whether the call originated from an untrusted class or not. We use the `checkTrustedDepth( )` method to help us with that: we pass it the stack depth number we mentioned earlier, and the string it should place in the exception if the test fails. Note that in the implementation of the `checkTrustedDepth( )` method that we have to add 1 to the class depth since calling this method has pushed another method frame onto the stack.

Otherwise, there are three interesting policies in this example: the network policy, the thread policy, and the file policy.

D.2.2.1 Implementing network access

Our implementation of the `checkConnect( )` method is designed to allow code to connect back to the host from which it was loaded. If there is no class loader on the stack, we want to permit access to any host, so we simply return. Otherwise, we obtain the hostname the untrusted class was loaded from (via the `getHost( )` method of the class loader) and compare that to the hostname the untrusted class is attempting to contact. If the strings are equal, we're all set and can return. Otherwise, we implement the logic we described earlier by obtaining the IP address for each hostname and comparing the two IP addresses.

If you choose to implement a different network security model for your `checkConnect( )` method, there are a few things that you should be aware of:

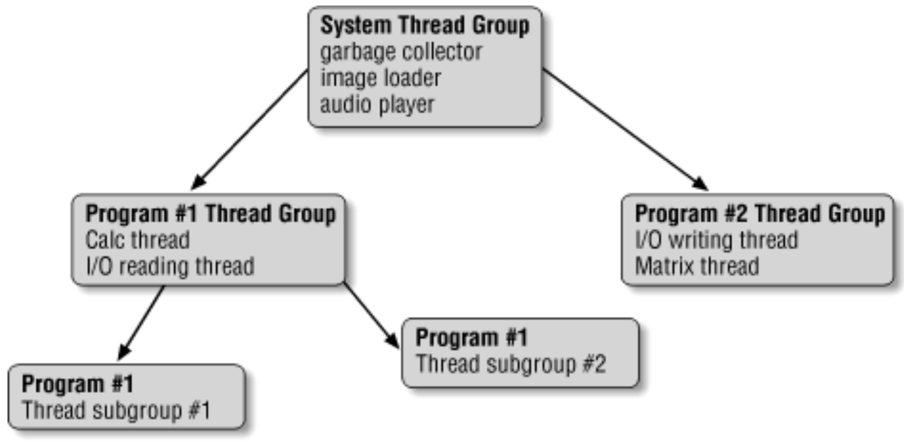
- The `checkConnect( )` method is frequently called with a port of −1. That usage comes primarily from the methods of the `InetAddress` class; in order to resolve the name of a machine, you must be able to make a connection to that machine. So if you want to restrict a connection to the privileged ports on your machine (those less than 1024), make sure you test to see that the port is between and 1023, rather than simply less than 1024.
- The host argument passed to the `checkConnect( )` method is frequently an IP address rather than a symbolic hostname. This is an artifact of the way in which the default socket implementation (that is, the `PlainSocketImpl` class) operates: this class actually generates two calls to the `checkConnect( )` method. The first call contains the actual hostname and a port number of −1 (because the `PlainSocketImpl` class has called the `InetAddress.getByName( )` method), and the second call contains the IP address and the actual port number.
- If you choose to disallow all network access by untrusted classes and you are using a network−based class loader to load classes, you cannot simply write a `checkConnect( )` method that calls the `inClassLoader( )` method and throws an exception if it returns `true`. The class loader must be allowed to open a socket in order to retrieve additional classes that are referenced by the untrusted class, and such a request will contain the untrusted class on the stack when the call is made. You can use the `inClass( )` method to see if the class loader is attempting to open the socket, in which case you should let the operation succeed.

You may want to implement a similar policy in the `checkAccept( )` method so that a class can only accept a connection from the host from which it was loaded.

D.2.2.2 Implementing thread security

Implementing a model of thread security requires that you implement the `checkAccess( )` methods as well as implementing the `getThreadGroup( )` method. Our example implements a hierarchical notion of thread permissions which fits well within the notion of the virtual machine's thread hierarchy (see Figure D−1). In this model, a thread can manipulate any thread that appears within its thread group or within a thread group that is descended from its thread group. In the example, Program #1 has created two thread groups. The Calc thread can manipulate itself, the I/O thread, and any thread in the Program #1 thread groups; it cannot manipulate any threads in the system thread group or in Program #2's thread group. Similarly, threads within Program #1's thread subgroup #1 can only manipulate threads within that group.

Figure D−1. A Java thread hierarchy



This is a different security model than that which is implemented by Java's `appletviewer` and by browsers in 1.1. In those models, any thread in any thread group of the applet can modify any other thread in any other thread group of the applet, but threads in one applet are still prevented from modifying threads in another applet or from modifying the system threads. But the model we'll describe fits the thread hierarchy a little better.

The key to our model of thread security depends on the `getThreadGroup( )` method. We can use this method to ensure that each class loader creates its threads in a new thread group as follows:

- If the program attempts to create a thread in a particular thread group, the `checkAccess( )` method can throw a security exception if the thread group in question is not a descendant of the thread group that belongs to the class loader.
- If the program attempts to create a thread without specifying the thread group to which it should belong, we can arrange for the `getThreadGroup( )` method to return the class loader's default thread group. This works because the constructors of the thread class call the `getThreadGroup( )` method directly to obtain the thread group to which a thread should belong.

The simplest way to implement `getThreadGroup( )` is to create a new thread group for each instance of a class loader. In a browser−type program, this does not necessarily create a new thread group for each applet because the same instance of a class loader might load two or more different applets if those applets share the same codebase. If we adopt this approach, those different applets will share the same default thread group. This might be considered a feature. It is also the approach we'll show; the necessary code to put different programs loaded by the same class loader into different thread groups is a straightforward extension.

We want each instance of a class loader to provide a different thread group. The simplest way to implement this logic is to defer to the class loader to provide the thread group. If there is no class loader, we'll use the thread group our superclass recommends (which, if we've directly extended the `SecurityManager` class, will be the thread group of the calling thread).

There are two caveats with this model. The first has to do with the way in which thread groups are created. When you create a thread group without specifying a parent thread group, the new thread group is placed into the thread hierarchy as a child of the thread group of the currently executing thread. For example, in Figure D−1, when the Calc thread creates a new thread group, by default that thread group is a child of Program Thread Group #1 (e.g., it could be Program Subgroup #1). Hence, if you start a program, you must ensure that it starts by executing it in the thread group that would be returned by its class loader −− that is, the default thread group of the program. We'll do that in our JavaRunner program.

The second caveat is that threads may not be expecting this type of enforcement of the thread hierarchy since it does not match many popular browser implementations. Hence, programs may fail under this model while they may succeed under a different model.

D.2.2.3 Implementing the file access methods

If you are going to implement a security manager, you must determine a policy for reading and writing files and implement it in each of the `checkRead( )` and `checkWrite( )` methods. The logic you put into each method is slightly different.

In the case where these methods take a single string argument, the logic is straightforward: the program is attempting to open a file with the given name, and you should either accept or reject that operation. We based our decision on the depth of the class loader since untrusted classes may not directly open a file for reading or writing, but they may cause that to happen through the Java API.

In the case where these methods take a `FileDescriptor` as an argument, the policy is a little harder to define. As far as the Java API is concerned, these methods are only called as a result of calling the `Socket.getInputStream( )` or `Socket.getOutputStream( )` methods −− which means that the security manager is really being asked to determine if the socket associated with the given file descriptor should be allowed to be read or written. By this time, the socket has already been created and has made a valid connection to the remote machine, and the security manager has had the opportunity to prohibit that connection at that time.

What type of access, then, would you prohibit when you implement these methods? It partially depends on the types of checks your security manager made when the socket was created. We'll assume for now that a socket created by an untrusted class can only connect to the site from which the class was loaded while a socket created by a trusted class can connect to any site. Hence, you might want to prohibit an untrusted class from opening the data stream of a socket created by a trusted class −− although if the class is trusted, you typically want to trust that class's judgment, and if that class passed the socket reference to an untrusted class, the untrusted class should be able to read from or write to the socket.

On the other hand, it is important to be sure that these methods are actually being called from the socket class. An untrusted class could attempt to pass an arbitrary file descriptor to the `File*Stream` constructor, breaking into your machine.

Typically, then, the only checks you put into this method are to determine that the `FileDescriptor` object is valid and the `FileDescriptor` object does indeed belong to the socket class.

D.3 Running Secure Applications

Now we can put this altogether and show the framework necessary to run the application. That framework is the following program:

```
package javasec.samples.appd;

import java.lang.reflect.*;

public class JavaRunner implements Runnable {
    final static int numArgs = 1;
    private JRClassLoader jrl;
    private Object args[];
    private String className;

    JavaRunner(JRClassLoader jrl, String className, Object args[]) {
        this.jrl = jrl;
        this.className = className;
        this.args = args;
    }

    void invokeMain(Class clazz) {
        Class argList[] = new Class[1];
        String strArray[] = new String[1];
        argList[0] = strArray.getClass(  );
        Method mainMethod = null;
        try {
            mainMethod = clazz.getMethod("main", argList);
        } catch (NoSuchMethodException nsme) {
            System.out.println("No main method in " + clazz.getName(  ));
            System.exit(-1);
        }

        try {
            mainMethod.invoke(null, args);
        } catch (Throwable e) {
            if (e instanceof InvocationTargetException)
                e = ((InvocationTargetException) e)
                            .getTargetException(  );
            System.out.println("Procedure exited with exception " + e);
            e.printStackTrace(  );
        }
    }

    public void run(  ) {
        Class target = null;
        try {
            target = jrl.loadClass(className);
            invokeMain(target);
        } catch (ClassNotFoundException cnfe) {
            System.out.println("Can't load " + className);
        }
    }

    static Object[] getArgs(String args[]) {
        String passArgs[] = new String[args.length - numArgs];
        for (int i = numArgs; i < args.length; i++)
            passArgs[i - numArgs] = args[i];

        Object wrapArgs[] = new Object[1];
        wrapArgs[0] = passArgs;
        return wrapArgs;
    }

    public static void main(String args[]) {
        System.setSecurityManager(new JavaRunnerManager(  ));
        JRClassLoader jrl = new JRClassLoader(args[0]);
        ThreadGroup tg = jrl.getThreadGroup(  );
        Thread t = new Thread(tg,
                new JavaRunner(jrl, args[1], getArgs(args)));
        t.start(  );
        try {
            t.join(  );
        } catch (InterruptedException ie) {
            System.out.println("Thread was interrupted");
        }
    }
}
```

The `main( )` method of this class installs our custom security manager, creates the class loader we showed earlier, creates a new thread group, and runs the target class in that new thread group. Object reflection is used to find the `main( )` method of the target class, which is passed any remaining arguments from the command line.

To run this program, the first argument is expected to be the URL from which the target application should be loaded. The second argument is the name of the target class, and any remaining arguments are passed to that class. Remember you may optionally define a trusted base from which trusted code can be loaded; so a command to run the class `Cat` through the `JavaRunner` might look like this:

```
piccolo% java -DtrustedBase=http://piccolo/trustedClasses/ \
                javasec.samples.appd.JavaRunner file:///files/sdo/classes/ \
                Cat /etc/hosts
```

This will attempt to load the `Cat` class (plus any classes it references) from the *trustedClasses* directory on piccolo's web server and then from the */files/sdo/classes* directory in the local filesystem.

D.4 Summary

The differences in the security models of Java 1.1 and Java 2 mean that running a security application in Java 1.1 requires a lot more work on the part of developers. The secure container we've created here allows you to run secure applications in Java 1.1 (and Java 2, for that matter); it follows the seeds of the security enhancements that are present in Java 2. However, it also serves as a good outline for a Java 2 container when you need to set new security managers or policies for programs you want to run.

Appendix E. Implementing a JCE Security Provider

When we developed engines in the examples throughout the book, we mentioned that the JCE engines must be deployed in a special way. In this appendix, we'll give the outline of those deployment rules and fill in the missing code required for JCE registration.

The steps we're outlining (along with parts of the code) can be found at http://java.sun.com/products/jce/doc/guide/HowToImplAProvider.html. We'll give a summary of that information here; for complete information, consult that URL.

Remember that the steps we list in this appendix are required only if you are implementing one of the following engines: `KeyGenerator`, `Mac`, `SecretKeyFactory`, `Cipher`, or `KeyAgreement`. If you want to develop other engines, you deploy the code for those engines just like any other Java code.

Here are the steps necessary to develop a JCE security provider.

1. Obtain a test JCE signing certificate.

   JCE security providers must be signed with a certificate issued by a special certificate authority. Both Sun and IBM are set up to provide these certificates. The application for such a certificate must be done both online and by mailing hardcopy documents; the resulting test certificate will be valid for 30 days.

   Part of what you must send includes a certificate signing request for a self−generated 1024−bit DSA public key. The `keytool` examples from Chapter 10, can be modified to produce such a request. What you'll eventually get back are two certificates: the root certificate of the JCE signing authority and your public key certificate, signed by the root certificate. Both certificates must be imported into a keystore for later use.
2. Write your security provider and engine classes.

   Based on the examples throughout this book, you should have a good idea of how to do that. However, JCE engines must perform special authentication when they are loaded; sample code to do that follows.
3. Package your provider and engine classes into a jar file, and use `jarsigner` along with your test JCE signing certificate to sign the jar file.
4. Test your implementation.
5. Once you've completed testing, apply for a full−fledged JCE signing certificate.
6. Sign the jar file containing your provider and engine classes with the new certificate, and ship the signed jar file.

Now we'll turn our attention to the code conventions that JCE engines must follow. All JCE engines must perform two things: they must verify that both the JCE framework classes and the engine itself were loaded from a jar file signed by a recognized JCE signing authority.

In our original security provider, we showed a `verifyForJCE( )` method that we'll use to perform this verification. The constructor of each of the JCE engines you develop must call this method (or execute similar code), which will throw a security exception if the classes cannot be verified.

The steps to verify each jar file are the same:

1. Find a reference to the jar file, using a class that is known to be in the jar file. For the JCE jar file, we'll use the class `javax.crypto.Cipher`. For your provider jar file, you should use the provider

class itself.
2. Ensure that each class contained in the jar file has a valid signature.
3. Ensure that the signer of the jar file was a trusted JCE signer.

   A trusted JCE signer is either one of the root JCE signers (Sun or IBM) or someone who has been issued a certificate from one of the root signers. Hence, your verification class must have one of those two certificates available.

   In fact, your verification class must embed one of those certificate into itself. It isn't sufficient to read the certificate from disk somewhere; someone could modify the certificate and then re−sign the jar file. So when you obtain the root certificate from Sun with your test certificate, you must embed that certificate into your code as explained below.

Here's a complete implementation of our provider that includes this code:

```
package javasec.samples.appe;

import java.net.*;
import java.io.*;
import java.util.jar.*;
import java.util.*;
import java.security.cert.*;
import java.security.PrivilegedAction;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.security.AccessController;
import java.security.CodeSource;
import java.security.Provider;

public class XYZProvider extends Provider {

    public XYZProvider(  ) {
        super("XYZ", 1.0, "XYZ Security Provider v1.0");
        put("KeyGenerator.XOR",
                    "javasec.samples.ch09.XORKeyGenerator");
        put("KeyPairGenerator.XYZ",
                    "javasec.samples.ch09.XYZKeyPairGenerator");
        put("KeyFactory.XYZ", "javasec.samples.ch09.XYZKeyFactory");
        put("MessageDigest.XYZ",
                    "javasec.samples.ch11.XYZMessageDigest");
        put("Signature.XYZwithSHA",
                    "javasec.samples.ch12.XYZSignature");
        put("Cipher.XOR", "javasec.samples.ch13.XORCipher");
        put("KeyManagerFactory.XYZ",
                    "javasec.samples.ch14.SSLKeyManagerFactory");

        // Now include any aliases
        put("Alg.Alias.MessageDigest.SHA−1", "SHA");
    }

    private static boolean verifiedJCE = false;
    private static X509Certificate[] trustedCerts;

    static byte[][] embeddedCerts = {
        /*
        { // Embed first certificate here
        },
        { // Embed second certificate here... and so on
        }
        */
    };

    static {
        // Note: This code is incomplete until you obtain the trusted
        // root certificate from Sun (or, optionally, IBM). When you
        // apply for your JCE signing certificate, they will send you
        // the root certificate. Then follow these steps:
        //    1) Import it into a keystore
        //    2) Write a program to read the certificate out of
        //       the keystore
        //    3) Get the encoded byte array from the certificate
        //    4) Dump out the byte array to a print writer file
        //       like this:
        //         for (int i = 0; i < bytearray.length; i++)
        //             pw.print(bytearray[i] + ", ");
        //    5) Convert that file into a byte array definition,
        //       and put it into the embeddedCerts array above

        try {
            trustedCerts = new X509Certificate[embeddedCerts.length];
            CertificateFactory cf =
                         CertificateFactory.getInstance("X509");
            for (int i = 0; i < trustedCerts.length; i++) {
                ByteArrayInputStream bais =
                        new ByteArrayInputStream(embeddedCerts[i]);
                trustedCerts[i] =
                    (X509Certificate) cf.generateCertificate(bais);
            }
        } catch (Exception e) {
            throw new SecurityException("Can't initialize certs " + e);
        }
    }

    public static final synchronized void verifyForJCE(  ) {
        // If the JCE has already been verified, just return.
        if (verifiedJCE)
            return;

        // We must verify both the JCE and our provider JAR files.
        // So we choose a class in each file.
        verify(javax.crypto.Cipher.class);
        verify(XYZProvider.class);
        verifiedJCE = true;
    }

    private static void verify(Class c) {
        // Verify that the class C comes from a jar file signed by a
        // trusted entity.

        // Find out the URL for the class
        final URL u = getURL(c);
        if (u == null)
            throw new SecurityException(
                           "Can't find valid signed class " + c);

        // Get the JAR file. The code that called us probably doesn't
        // have permission to read the local jar file, so we must do
        // this as a privileged block.
        JarFile jf;
        try {
            jf = (JarFile) AccessController.doPrivileged(
                new PrivilegedExceptionAction(  ) {
                    public Object run(  ) throws Exception {
```

```
                    return ((JarURLConnection)
                                u.openConnection()).getJarFile(  );
                }
            }
        );
    } catch (PrivilegedActionException pae) {
        throw new SecurityException(
                         "Cannot authenticate JCE " + pae);
    }

    try {
        verifySingleJarFile(jf);
    } catch (Exception e) {
        throw new SecurityException(
                         "Cannot authenticate JCE " + e);
    }
}

// Verify that a jar file is signed.
private static void verifySingleJarFile(JarFile jf)
                    throws IOException, CertificateException {
    Vector entriesVec = new Vector(  );

    // Ensure there is a manifest file
    Manifest man = jf.getManifest(  );
    if (man == null)
        throw new SecurityException(
                         "The JCE framework is not signed");

    // Ensure that all signed entries are signed correctly
    byte[] buffer = new byte[8192];
    Enumeration entries = jf.entries(  );
    while (entries.hasMoreElements(  )) {
        JarEntry je = (JarEntry)entries.nextElement(  );
        entriesVec.addElement(je);
        InputStream is = jf.getInputStream(je);
        int n;
        while ((n = is.read(buffer, 0, buffer.length)) != -1) {
            // Just read; the JarInputStream will do the signature
            // verification automatically. If the signature is
            // invalid, the read will throw a security exception.
        }
        is.close(  );
    }
    jf.close(  );

    // We know now that if the entry in the jar file was signed,
    // the signature was valid. Now make sure that every entry
    // was signed.
    // First, get the list of signer certificates from the jar file
    Enumeration e = entriesVec.elements(  );
    while (e.hasMoreElements(  )) {
        JarEntry je = (JarEntry) e.nextElement(  );
        if (je.isDirectory(  ))
            continue;
        // Every file must be signed - except files in META-INF
        Certificate[] certs = je.getCertificates(  );
        if ((certs == null) || (certs.length == 0)) {
            if (!je.getName(  ).startsWith("META-INF"))
                throw new SecurityException(
                            "Unsigned JCE classes!");
            else ;
        } else {
            // Check whether the file is signed by the trusted
            // JCE authority. This is the case if somewhere along
            // the chain of certificates we find the trusted
            // signer.
            Certificate[] chainRoots = getChainRoots(certs);
            boolean signedAsExpected = false;
            for (int i = 0; i < chainRoots.length; i++) {
                if (isTrusted((X509Certificate)chainRoots[i])) {
                    signedAsExpected = true;
                    break;
                }
            }
            if (!signedAsExpected)
                throw new SecurityException("The JCE framework " +
                              "is not signed by a trusted signer");
        }
    }
}

private static boolean isTrusted(X509Certificate cert) {
    // Return true only if either of the following is true:
    // 1) the cert is in the trustedCerts.
    // 2) the cert is issued by a trusted CA.
    for (int i = 0; i < trustedCerts.length; i++) {
        // If the cert has the same SubjectDN as a trusted CA,
        // check whether the two certs are the same.
        if (cert.getSubjectDN(  ).equals(
                            trustedCerts[i].getSubjectDN(  ))) {
            if (cert.equals(trustedCerts[i]))
                return true;
        }
    }

    // Check whether the cert is issued by a trusted CA.
    // Signature verification is expensive. So we check
    // whether the cert is issued by one of the trusted CAs
    // only if the above loop failed.
    for (int i = 0; i < trustedCerts.length; i++) {
        // If the issuer of the cert has the same name as
        // a trusted CA, check whether that trusted CA
        // actually issued the cert.
        if (cert.getIssuerDN(  ).equals(
                           trustedCerts[i].getSubjectDN(  ))) {
            try {
                cert.verify(trustedCerts[i].getPublicKey(  ));
                return true;
            } catch (Exception e) {
                // That's okay; try the next one.
            }
        }
     }
     return false;
 }

 private static Certificate[] getChainRoots(Certificate[] certs) {
     Vector result = new Vector(3);
     // Choose a Vector size that seems reasonable
     for (int i = 0; i < certs.length - 1; i++) {
         if (!((X509Certificate)certs[i + 1]).
               getSubjectDN(  ).equals(
             ((X509Certificate)certs[i]).getIssuerDN(  ))) {
             // We've reached the end of a chain
             result.addElement(certs[i]);
            }
        }
        // The final entry in the certs array is always
        // a "root" certificate
        result.addElement(certs[certs.length - 1]);
        Certificate[] ret = new Certificate[result.size(  )];
        result.copyInto(ret);

        return ret;
    }

    /*
     * Returns the URL of the JAR file containing the specified class.
     */
     private static URL getURL(Class c) {
        final Class cc = c;
        return (URL)AccessController.doPrivileged(
            new PrivilegedAction(  ) {
                public Object run(  ) {
                    CodeSource s1 =
                          cc.getProtectionDomain().getCodeSource(  );
                    return s1.getLocation(  );
                }
             }
         );
      }
}
```

Appendix F. Quick Reference

This appendix contains a quick−reference guide to the classes that we have discussed in this book. The primary focus is on classes in the core `java.security` package, as well as the packages that come with the three security extensions: `javax.crypto`, `javax.net`, `javax.security`, and `com.net.sun.ssl`. Accordingly, the classes listed in this appendix are organized by their primary package. Of course, there are a number of security−related classes −− such as the various permission classes −− that do not belong to one of these packages; these are listed in Section F.19 at the end of this appendix. Information in this appendix is based only on Java 2, version 1.3.

A note about class names: class names appear unqualified (without their package names). The exception to this is the various `Certificate` classes and interfaces. When `Certificate` appears without a package name, it refers to the `java.security.cert.Certificate` class. Otherwise, it will be fully−qualified (e.g., `javax.security.cert.Certificate`). The same is true of `X509Certificate`.

The `Policy` class, if unqualified, refers to the `java.security.Policy` class.

F.1 Package java.security

Class java.security.AccessControlContext

An access control context allows the access controller to substitute a different context (that is, a different set of protection domains) than the context provided by the stack of the current thread. This class might be used by a server thread to determine if a particular calling thread should be allowed to perform particular operations.

Class Definition

```
public final class java.security.AccessControlContext
    extends java.lang.Object {

    // Constructors
    public AccessControlContext(AccessControlContext, DomainCombiner);
    public AccessControlContext(ProtectionDomain[]);

    // Instance Methods
    public void checkPermission(Permission);
    public boolean equals(Object);
    public DomainCombiner getDomainCombiner(  );
    public int hashCode(  );
}
```

See also

`AccessController`

Class java.security.AccessController

The access controller is responsible for determining whether or not the current thread can execute a given operation. This decision occurs in the `checkPermission( )` method and is based upon all the protection domains that are on the stack of the calling thread and the set of permissions that have been granted to those protection domains. The access controller is heavily used by the security manager to enforce a specific security policy, and it may be used by arbitrary code to enforce an application−specific security policy as well.

Class Definition

```
public final class java.security.AccessController
    extends java.lang.Object {

    // Class Methods
    public static void checkPermission(Permission);
    public static native Object
                  doPrivileged(PrivilegedExceptionAction);
    public static native Object
                  doPrivileged(PrivilegedAction, AccessControlContext);
    public static native Object doPrivileged(PrivilegedAction);
    public static native Object doPrivileged(
                   PrivilegedExceptionAction, AccessControlContext);
    public static AccessControlContext getContext(  );
}
```

See also

`Permission`, *`ProtectionDomain`*, *`Policy`*

Class java.security.AlgorithmParameterGenerator

This engine class is used to generate algorithm−specific parameters, which may then be turned into algorithm parameter specifications to be used to initialize other engine classes. In normal usage, those engines can be initialized directly via the same `init( )` methods that exist in this class; hence, this class is seldom used.

Class Definition

```
public class java.security.AlgorithmParameterGenerator {

        // Constructors
        protected AlgorithmParameterGenerator(
                          AlgorithmParameterGeneratorSpi, Provider, String);

        // Class Methods
        public static final AlgorithmParameterGenerator
                                           getInstance(String);
        public static final AlgorithmParameterGenerator
                                           getInstance(String, String);

        // Instance Methods
        public final String getAlgorithm(  );
        public final Provider getProvider(  );
        public final void init(int);
        public final void init(int, SecureRandom);
        public final void init(AlgorithmParameterSpec);
        public final void init(AlgorithmParameterSpec, SecureRandom);
        public final AlgorithmParameters generateParameters(  );
}
```

See also

AlgorithmParameters

Class java.security.AlgorithmParameter−GeneratorSpi

This class is the service provider interface for the algorithm parameter generator. If you want to implement your own algorithm parameter generator, you subclass this class and register your implementation with an appropriate security provider.

Class Definition

```
public abstract class java.security.AlgorithmParameterGeneratorSpi {

        // Instance Methods
        protected abstract void engineInit(int, SecureRandom);
        protected abstract void engineInit(
                                      AlgorithmParameterSpec, SecureRandom);
        protected abstract AlgorithmParameters engineGenerateParameters(  );
}
```

See also

AlgorithmParameterGenerator

Class java.security.AlgorithmParameters

This engine class is used to generate algorithm−specific parameter specifications, which may then be used to initialize other engine classes. In normal usage, those engines can be initialized directly via the same `init( )` methods that exist in this class; hence, this class is seldom used.

Class Definition

```
public class java.security.AlgorithmParameters {

        // Class Methods
        public static final AlgorithmParameters getInstance(String);
        public static final AlgorithmParameters getInstance(
                                           String, String);

        // Constructors
        protected AlgorithmParameters(AlgorithmParametersSpi,
                                      Provider, String);

        // Instance Methods
        public final String getAlgorithm(  );
        public final Provider getProvider(  );
        public final void init(AlgorithmParameterSpec);
        public final void init(byte[]);
        public final void init(byte[], String);
        public final AlgorithmParameterSpec getParameterSpec(Class);
        public final byte[] getEncoded(  );
        public final byte[] getEncoded(String);
        public final String toString(  );
}
```

See also

KeyPairGenerator

Class java.security.AlgorithmParametersSpi

This is the service provider interface for algorithm parameters. If you want to implement your own algorithm parameters, you do so by subclassing this class and registering your implementation with an appropriate security provider.

Class Definition

```
public abstract class java.security.AlgorithmParametersSpi
                extends java.lang.Object {

        // Constructors
        public AlgorithmParametersSpi(  );

        // Protected Instance Methods
        protected abstract byte[] engineGetEncoded(  );
        protected abstract byte[] engineGetEncoded(String);
        protected abstract AlgorithmParameterSpec
                                   engineGetParameterSpec(Class);
        protected abstract void engineInit(AlgorithmParameterSpec);
        protected abstract void engineInit(byte[]);
        protected abstract void engineInit(byte[], String);
        protected abstract String engineToString(  );
}
```

See also

AlgorithmParameters

Class java.security.AllPermission

This class represents permissions to perform any operation. This permission is typically granted to extension classes, which (like the core API) need to be able to perform any operation. Although it is a permission class, instances of this class have no name and no actions. The `implies( )` method of this class always returns `true`.

Class Definition

```
public final class java.security.AllPermission
        extends java.security.Permission {

        // Constructors
        public AllPermission(  );
        public AllPermission(String, String);

        // Instance Methods
        public boolean equals(Object);
        public String getActions(  );
        public int hashCode(  );
        public boolean implies(Permission);
        public PermissionCollection newPermissionCollection(  );
}
```

See also

Permission

Class java.security.BasicPermission

A basic permission represents a binary permission −− that is, a permission that you either have or do not have. Hence, the action string in a basic permission is unused. A basic permission follows the same naming convention as java properties: a series of period−separated words, like "exitVM" or "xyz.payrollPermission". The `BasicPermission` class is capable of wildcard matching if the last word in the permission is an asterisk. This class serves as the superclass for a number of default permission classes.

Class Definition

```
public abstract class java.security.BasicPermission
        extends java.security.Permission
        implements java.io.Serializable {

        // Constructors
        public BasicPermission(String);
        public BasicPermission(String, String);

        // Instance Methods
        public boolean equals(Object);
        public String getActions(  );
        public int hashCode(  );
        public boolean implies(Permission);
        public PermissionCollection newPermissionCollection(  );
}
```

See also

Permission, PermissionCollection, SecurityPermission, java.awt.AwtPermission, java.io.SerializablePermission, java.lang.RuntimePermission, java.lang.reflect.ReflectPermission, java.net.NetPermission, java.sql.SqlPermission, java.util.PropertyPermission, javax.security.auth.AuthPermission

Interface java.security.Certificate

Classes that implement this interface can be used to model certificates. However, this interface has been deprecated in favor of the `java.security.cert.Certificate` class.

```
public interface java.security.Certificate {

    // Instance Methods
    public abstract void decode(InputStream);
    public abstract void encode(OutputStream);
    public abstract String getFormat(  );
    public abstract Principal getGuarantor(  );
    public abstract Principal getPrincipal(  );
    public abstract PublicKey getPublicKey(  );
    public abstract String toString(boolean);
}
```

See also

`java.security.cert.Certificate`

Class java.security.CodeSource

A code source encapsulates the location from which a particular class was loaded and the public keys (if any) that were used to sign the class. This information is used by a secure class loader to define a protection domain associated with the class; typically, the class loader is the only object that uses a code source.

Class Definition

```
public class java.security.CodeSource
        extends java.lang.Object
        implements java.io.Serializable {

        // Constructors
        public CodeSource(URL, Certificate[]);

        // Instance Methods
        public boolean equals(Object);
        public final Certificate[] getCertificates(  );
        public boolean implies(  );
        public final URL getLocation(  );
        public int hashCode(  );
        public String toString(  );
}
```

See also

`SecureClassLoader`, *`ProtectionDomain`*

Class java.security.DigestInputStream

A digest input stream is an input filter stream that is associated with a message digest object. As data is read from the input stream, it is automatically passed to its associated message digest object; once all the data has been read, the message digest object will return the hash of the input data. You must have an existing input stream and an initialized message digest object to construct this class; once the data has passed through the stream, call the methods of the message digest object explicitly to obtain the hash.

Class Definition

```
public class java.security.DigestInputStream
        extends java.io.FilterInputStream {

        // Variables
        protected MessageDigest digest;

        // Constructors
        public DigestInputStream(InputStream, MessageDigest);

        // Instance Methods
        public MessageDigest getMessageDigest(  );
        public void on(boolean);
        public int read(  );
        public int read(byte[], int, int);
        public void setMessageDigest(MessageDigest);
        public String toString(  );
}
```

See also

`DigestOutputStream`, *`MessageDigest`*

Class java.security.DigestOutputStream

A digest output stream is a filter output stream that is associated with a message digest object. When data is written to the output stream, it is also passed to the message digest object so that when the data has all been written to the output stream, the hash of that data may be obtained from the digest object. You must have an existing output stream and an initialized message digest object to use this class.

Class Definition

```
public class java.security.DigestOutputStream
        extends java.io.FilterOutputStream {

        // Variables
        protected MessageDigest digest;

        // Constructors
        public DigestOutputStream(OutputStream, MessageDigest);

        // Instance Methods
        public MessageDigest getMessageDigest(  );
        public void on(boolean);
        public void setMessageDigest(MessageDigest);
        public String toString(  );
        public void write(int);
        public void write(byte[], int, int);
}
```

See also

`DigestInputStream`, *`MessageDigest`*

Interface java.security.DomainCombiner

Classes that implement this interface are used by the access controller to optimize domain checking by combining permissions from domains. It is normally an opaque object to developers.

Class definition

```
public interface java.security.DomainCombiner {

    // Instance Methods
    public abstract ProtectionDomain[]
                    combine(ProtectionDomain[], ProtectionDomain[]);
}
```

See also

`javax.security.auth.SubjectDomainCombiner`

Interface java.security.Guard

An object of a class that implements the `Guard` interface may be used to protect access to a resource. In typical usage, a guard is an object of the `Permission` class, so that access to the guarded resource is granted if and only if the current thread has been granted the given permission. This interface is used by the `GuardedObject` class to guard access to another object.

Interface Definition

```
public interface java.security.Guard {

        // Instance Methods
        public abstract void checkGuard(Object);
}
```

See also

`GuardedObject`, *`Permission`*

Class java.security.GuardedObject

A guarded object is a container for another object. The contained object is guarded using an object that implements the `Guard` interface; in typical usage, that would be an instance of a `Permission` object. The guarded object stores a serialized version of the object it contains; the contained object will be deserialized and returned by the `getObject( )` method only if the guarded object allows access.

Class Definition

```
public class java.security.GuardedObject
        extends java.lang.Object
        implements java.io.Serializable {

        // Constructors
        public GuardedObject(Serializable, Guard);

        // Instance Methods
        public Object getObject(  );
}
```

See also

`Guard`

Class java.security.Identity

An identity encapsulates public knowledge about an entity (that is, a person or a corporation −− or anything that could hold a public key). Identities have names and may hold a public key, along with a certificate chain to validate the public key. An identity may belong to an identity scope, but this feature is optional and is not typically used. This class is deprecated.

Class Definition

```
public abstract class java.security.Identity
        extends java.lang.Object
        implements java.security.Principal, java.io.Serializable {

        // Constructors
        protected Identity(  );
        public Identity(String);
        public Identity(String, IdentityScope);

        // Instance Methods
        public void addCertificate(java.security.Certificate);
        public final boolean equals(Object);
        public java.security.Certificate[] certificates(  );
        public String getInfo(  );
        public final String getName(  );
        public PublicKey getPublicKey(  );
        public final IdentityScope getScope(  );
        public int hashCode(  );
        public void removeCertificate(java.security.Certificate);
        public void setInfo(String);
        public void setPublicKey(PublicKey);
        public String toString(  );
        public String toString(boolean);

        // Protected Instance Methods
        protected boolean identityEquals(Identity);
}
```

See also

java.security.Certificate, *IdentityScope*, *Principal*, *PublicKey*

Class java.security.IdentityScope

An identity scope is a collection of identities; an identity may belong to a single identity scope. The notion is that scope is recursive: an identity scope may itself belong to another identity scope (or it may be unscoped). This class is deprecated in Java 2.

Class Definition

```
public abstract class java.security.IdentityScope
        extends java.security.Identity {

        // Constructors
        protected IdentityScope(  );
        public IdentityScope(String);
        public IdentityScope(String, IdentityScope);

        // Class Methods
        public static IdentityScope getSystemScope(  );
        protected static void setSystemScope(IdentityScope);

        // Instance Methods
        public abstract void addIdentity(Identity);
        public abstract Identity getIdentity(String);
        public Identity getIdentity(Principal);
        public abstract Identity getIdentity(PublicKey);
        public abstract Enumeration identities(  );
        public abstract void removeIdentity(Identity);
        public abstract int size(  );
        public String toString(  );
}
```

See also

Identity

Interface java.security.Key

A key is essentially a series of bytes that are used by a cryptographic algorithm. Depending on the type of the key, the key may be used only for particular operations and only for particular algorithms, and it may have certain mathematical properties (including a mathematical relationship to other keys). The series of bytes that comprise a key is the encoded format of the key.

Interface Definition

```
public interface java.security.Key
        implements java.io.Serializable {

        // Instance Methods
        public abstract String getAlgorithm(  );
        public abstract byte[] getEncoded(  );
        public abstract String getFormat(  );
}
```

See also

`PrivateKey`, *`PublicKey`*, *`javax.crypto.SecretKey`*

Class java.security.KeyFactory

A key factory is an engine class that is capable of translating between public or private key objects and their external format (and vice versa). Hence, key factories may be used to import or export keys, as well as to translate keys of one class (e.g., `com.acme.DSAPublicKey`) to another class (e.g., `com.xyz.DSAPublicKeyImpl`) as long as those classes share the same base class. Key factories operate in terms of key specifications; these specifications are the various external formats in which a key may be transmitted. Keys are imported via the `generatePublic( )` and `generatePrivate( )` methods, they are exported via the `getKeySpec( )` method, and they are translated via the `translateKey( )` method.

Class Definition

```
public class java.security.KeyFactory
        extends java.lang.Object {

        // Constructors
        protected KeyFactory(KeyFactorySpi, Provider, String);

        // Class Methods
        public static final KeyFactory getInstance(String);
        public static final KeyFactory getInstance(String, String);

        // Instance Methods
        public final PrivateKey generatePrivate(KeySpec);
        public final PublicKey generatePublic(KeySpec);
        public final String getAlgorithm(  );
        public final KeySpec getKeySpec(Key, Class);
        public final Provider getProvider(  );
        public final Key translateKey(Key);
}
```

See also

`KeyFactorySpi`, *`KeySpec`*

Class java.security.KeyFactorySpi

This is the service provider interface for a key factory; if you want to implement your own key factory, you do so by extending this class and registering your implementation with an appropriate security provider. Instances of this class are expected to know how to create key objects from external key specifications and vice versa.

Class Definition

```
public abstract class java.security.KeyFactorySpi
        extends java.lang.Object {

        // Constructors
        public KeyFactorySpi(  );

        // Protected Instance Methods
        protected abstract PrivateKey engineGeneratePrivate(KeySpec);
        protected abstract PublicKey engineGeneratePublic(KeySpec);
        protected abstract KeySpec engineGetKeySpec(Key, Class);
        protected abstract Key engineTranslateKey(Key);
}
```

See also

`KeyFactory`, *`KeySpec`*

Class java.security.KeyPair

Public and private keys are mathematically related to each other and hence are generated together; this class provides an encapsulation of both the keys as a convenience to key generation.

Class Definition

```
public final class java.security.KeyPair
        extends java.lang.Object {

        // Constructors
        public KeyPair(PublicKey, PrivateKey);

        // Instance Methods
        public PrivateKey getPrivate(  );
        public PublicKey getPublic(  );
}
```

See also

`KeyPairGenerator`, *`PrivateKey`*, *`PublicKey`*

Class KeyPairGenerator

This is an engine class that is capable of generating a public key and its related private key. Instances of this class will generate key pairs that are appropriate for a particular algorithm (DSA, RSA, etc.). A key pair generator may be initialized to return keys of a particular strength (which is usually the number of bits in the key), or it may be initialized in an algorithmic−specific way; the former case is the one implemented by most key generators. An instance of this class may be used to generate any number of key pairs.

Class Definition

```
public abstract class java.security.KeyPairGenerator
        extends java.security.KeyPairGeneratorSpi {

        // Constructors
        protected KeyPairGenerator(String);

        // Class Methods
        public static KeyPairGenerator getInstance(String);
        public static KeyPairGenerator getInstance(String, String);

        // Instance Methods
        public final KeyPair genKeyPair(  );
        public KeyPair generateKeyPair(  );
        public String getAlgorithm(  );
        public final Provider getProvider(  );
        public void initialize(int);
        public void initialize(int, SecureRandom)
        public void initialize(AlgorithmParameterSpec, SecureRandom);
        public void initialize(AlgorithmParameterSpec);
}
```

See also

AlgorithmParameterSpec, *KeyPair*

Class KeyPairGeneratorSpi

This is the service provider interface class for the key pair generation engine; if you want to implement your own key pair generator, you must extend this class and register your implementation with an appropriate security provider. Instances of this class must be prepared to generate key pairs of a particular strength (or length); they may optionally accept an algorithmic−specific set of initialization values.

Class Definition

```
public abstract class java.security.KeyPairGeneratorSpi
        extends java.lang.Object {

        // Constructors
        public KeyPairGeneratorSpi(  );

        // Instance Methods
        public abstract KeyPair generateKeyPair(  );
        public abstract void initialize(int, SecureRandom);
        public void initialize(AlgorithmParameterSpec, SecureRandom);
}
```

See also

`AlgorithmParameterSpec`, *`KeyPairGenerator`*, *`SecureRandom`*

Class java.security.KeyStore

This class is responsible for maintaining a set of keys and their related owners. In the default implementation, this class maintains the *.keystore* file held in the user's home directory, but you may provide an alternate implementation of this class that holds keys anywhere: in a database, on a remote filesystem, on a Java smart card, or any and all of the above. The class that is used to provide the default keystore implementation is specified by the `keystore` property in the *$JDKHOME/lib/java.security* file. The keystore may optionally require a passphrase for access to the entire keystore (via the `load( )` method); this passphrase is often used only for sanity checking and is often not specified at all. On the other hand, private keys in the keystore should be protected (e.g., encrypted) by using a different passphrase for each private key.

Note that although the keystore associates entities with keys, it does not rely upon the `Identity` class itself.

Class Definition

```
public abstract class java.security.KeyStore
        extends java.lang.Object {

        // Constructors
        protected KeyStore(KeyStoreSpi, Provider, String);

        // Class Methods
        public static final String getDefaultType(  );
        public static KeyStore getInstance(String);
        public static KeyStore getInstance(String, String);

        // Instance Methods
        public final Enumeration aliases(  );
        public final boolean containsAlias(String);
        public final void deleteEntry(String);
        public final Certificate getCertificate(String);
        public final String getCertificateAlias(Certificate);
        public final Certificate[] getCertificateChain(String);
        public final Date getCreationDate(String);
        public final Key getKey(String, char[]);
        public final Provider getProvider(  );
        public final String getType(  );
        public final boolean isCertificateEntry(String);
        public final boolean isKeyEntry(String);
        public final void load(InputStream, char[]);
        public final void setCertificateEntry(String, Certificate);
        public final void setKeyEntry(String, Key, char[], Certificate[]);
        public final void setKeyEntry(String, byte[], Certificate[]);
        public final int size(  );
        public final void store(OutputStream, char[]);
}
```

See also

Certificate, *PublicKey*, *PrivateKey*, *SecretKey*

Class java.security.KeyStoreSpi

This is the service provider interface for implementations of the keystore engine. Each method of the `KeyStore` class corresponds naturally to a method of this engine, which is expected to provide the required information. When the engine is asked to store a key and is given a password (an array of characters), the engine is expected to encrypt the key. The engine should not encrypt the entire database (using the password to the `store( )` method); the database password is used to detect tampering but cannot be required to read the keystore.

Class Definition

```
public abstract class java.security.KeyStoreSpi
    extends java.lang.Object {

    // Constructors
    public KeyStoreSpi(  );

    // Instance Methods
    public abstract Enumeration engineAliases(  );
    public abstract boolean engineContainsAlias(String);
    public abstract void engineDeleteEntry(String);
    public abstract Certificate engineGetCertificate(String);
    public abstract String engineGetCertificateAlias(Certificate);
    public abstract Certificate[] engineGetCertificateChain(String);
    public abstract Date engineGetCreationDate(String);
    public abstract Key engineGetKey(String, char[]);
    public abstract boolean engineIsCertificateEntry(String);
    public abstract boolean engineIsKeyEntry(String);
    public abstract void engineLoad(InputStream, char[]);
    public abstract void engineSetCertificateEntry(
                               String, Certificate);
    public abstract void engineSetKeyEntry(String, byte[],
                               Certificate[]);
    public abstract void engineSetKeyEntry(String, Key, char[],
                                Certificate[]);
    public abstract int engineSize(  );
    public abstract void engineStore(OutputStream, char[]);
}
```

See also

KeyStore

Class java.security.MessageDigest

The message digest class is an engine class that can produce a one−way hash value for any arbitrary input. Message digests have two properties: they produce a unique hash for each set of input data (subject to the number of bits that are output), and the original input data is indiscernible from the hash output. The hash value is variously called a digital fingerprint or a digest. Message digests are components of digital signatures, but they are useful in their own right to verify that a set of data has not been corrupted. Once a digest object is created, data may be fed to it via the `update( )` methods; the hash itself is returned via the `digest( )` method.

Class Definition

```
public abstract class java.security.MessageDigest
        extends java.security.MessageDigestSpi {

        // Constructors
        protected MessageDigest(String);

        // Class Methods
        public static MessageDigest getInstance(String);
        public static MessageDigest getInstance(String, String);
        public static boolean isEqual(byte[], byte[]);

        // Instance Methods
        public Object clone(  );
        public byte[] digest(  );
        public byte[] digest(byte[]);
        public int digest(byte[], int, int);
        public final String getAlgorithm(  );
        public final int getDigestLength(  );
        public final Provider getProvider(  );
        public void reset(  );
        public String toString(  );
        public void update(byte);
        public void update(byte[]);
        public void update(byte[], int, int);
}
```

See also

`javax.crypto.Mac`

Class java.security.MessageDigestSpi

This is the service provider interface for the message digest engine; if you want to implement your own message digest class, you do so by extending this class and registering your implementation with an appropriate security provider. Since the `MessageDigest` class itself extends this class, you may also extend the `MessageDigest` class directly. Implementations of this class are expected to accumulate a hash value over data that is fed to it as a series of arbitrary bytes.

Class Definition

```
public abstract class java.security.MessageDigestSpi
        extends java.lang.Object {

        // Constructors
        public MessageDigestSpi(  );

        // Instance Methods
        public Object clone(  );

        // Protected Instance Methods
        protected abstract byte[] engineDigest(  );
        protected int engineDigest(byte[], int, int);
        protected int engineGetDigestLength(  );
        protected abstract void engineReset(  );
        protected abstract void engineUpdate(byte);
        protected abstract void engineUpdate(byte[], int, int);
}
```

See also

MessageDigest

Class java.security.Permission

This class forms the base class for all types of permissions that are used by the access controller. A permission object encapsulates a particular operation (e.g., reading the file */tmp/foo*). It does not, however, grant permission for that operation; rather, the permission object is constructed and passed to the access controller to see if that operation is one which the current security policy has defined as a permissible operation.

Permissions have names (e.g., the name of the file, or the name of the operation) and may optionally have actions (the semantics of which are dependent upon the type of permission). It is up to the `implies( )` method to determine if one permission grants another; this allows you to specify wildcard−type permissions that imply specific permissions (e.g., the permission named "*" may imply the permission named "myfile").

Class Definition

```
public abstract class java.security.Permission
        extends java.lang.Object
        implements java.security.Guard, java.io.Serializable {

        // Constructors
        public Permission(String);

        // Instance Methods
        public void checkGuard(Object);
        public abstract boolean equals(Object);
        public abstract String getActions(  );
        public final String getName(  );
        public abstract int hashCode(  );
        public abstract boolean isReadOnly(  );
        public void setReadOnly(  );
        public PermissionCollection newPermissionCollection(  );
        public String toString(  );
}
```

See also

AccessController, AllPermission, BasicPermission, FilePermission, Guard, PermissionCollection, Policy, SocketPermission

Class java.security.PermissionCollection

As you might infer, a permission collection is a collection of permission objects. In theory, a permission collection can be a set of arbitrary, unrelated permission objects; however, that usage is best avoided and left to the `Permissions` class. Hence, a permission collection should be thought of as a collection of one type of permission: a set of file permissions, a set of socket permissions, etc. A permission collection is responsible for determining if an individual permission (passed as a parameter to the `implies( )` method) is contained in the set of permissions in the object; presumably, it will do that more efficiently than by calling the `implies( )` method on each permission in the collection. If you implement a new permission class that has wildcard semantics for its names, then you must implement a corresponding permission collection to aggregate instances of that class (if you don't need wildcard matching, the default implementation of the `Permission` class will provide an appropriate collection).

Class Definition

```
public abstract class java.security.PermissionCollection
        extends java.lang.Object
        implements java.io.Serializable {

        // Constructors
        public PermissionCollection(  );

        // Instance Methods
        public abstract void add(Permission);
        public abstract Enumeration elements(  );
        public abstract boolean implies(Permission);
        public boolean isReadOnly(  );
        public void setReadOnly(  );
        public String toString(  );
}
```

See also

`Permission`, *`Permissions`*

Class java.security.Permissions

This class is an aggregate of permission collections. Hence, it is an appropriate collection object for a group of unrelated permissions, which is its typical use: the `Policy` class uses instances of this class to represent all the permissions associated with a particular protection domain.

Class Definition

```
public final class java.security.Permissions
        extends java.security.PermissionCollection
        implements java.io.Serializable {

        // Constructors
        public Permissions(  );

        // Instance Methods
        public void add(Permission);
        public Enumeration elements(  );
        public boolean implies(Permission);
}
```

See also

`Permission`, *`PermissionCollection`*, *`Policy`*

Class java.security.Policy

The `Policy` class encapsulates all the specific permissions that the virtual machine knows about. This set of permissions is by default read from a series of URLs specified by `policy.url` properties in the *$HOME/lib/security/java.security* file, although applications may specify their own policy objects by using the `setPolicy( )` method of this class. Alternately, a different default implementation of the policy class may be specified by changing the `policy.provider` property in the *java.security* file.

Class Definition

```
public abstract class java.security.Policy
        extends java.lang.Object {

        // Constructors
        public Policy(  );

        // Class Methods
        public static Policy getPolicy(  );
        public static void setPolicy(Policy);

        // Instance Methods
        public abstract PermissionCollection getPermissions(CodeSource);
        public abstract void refresh(  );
}
```

See also

`Permission`, *`Permissions`*

Interface java.security.Principal

A principal is anything that has a name, such as an identity. The name in this case is often an X.500 distinguished name, but that is not a requirement.

Interface Definition

```
public interface java.security.Principal {

        // Instance Methods
        public abstract boolean equals(Object);
        public abstract String getName(  );
        public abstract int hashCode(  );
        public abstract String toString(  );
}
```

See also

Identity

Interface java.security.PrivateKey

A private key is a key with certain mathematical properties that allows it to perform inverse cryptographic operations with its matching public key. Classes implement this interface only for type identification.

Interface Definition

```
public interface java.security.PrivateKey
        implements java.security.Key {
}
```

See also

Key, *PublicKey*

Class java.security.ProtectionDomain

A protection domain encapsulates the location from which a class was loaded and the keys used to sign the class (that is, a `CodeSource` object) and the set of permissions that should be granted to that class. These protection domains are consulted by the access controller to determine if a particular operation should succeed; if the operation is in the set of permissions in each protection domain on the stack, then the operation will succeed. This class is typically only used within a class loader.

Class Definition

```
public class java.security.ProtectionDomain
        extends java.lang.Object {

        // Constructors
        public ProtectionDomain(CodeSource, PermissionCollection);

        // Instance Methods
        public final CodeSource getCodeSource(  );
        public final PermissionCollection getPermissions(  );
        public boolean implies(Permission);
        public String toString(  );
}
```

See also

AccessController, *CodeSource*, *Permissions*

Class java.security.Provider

An instance of the `Provider` class is responsible for mapping particular implementations to desired algorithm/engine pairs; instances of this class are consulted (indirectly) by the `getInstance( )` methods of the engine classes to find a class that implements the desired operation. Instances of this class must be registered either with the `Security` class or by listing them in the *$JDKHOME/lib/security/java.security* file as a `security.provider` property.

Class Definition

```
public abstract class java.security.Provider
        extends java.util.Properties {

        // Constructors
        protected Provider(String, double, String);

        // Instance Methods
        public synchronized void clear(  );
        public Set entrySet(  );
        public String getInfo(  );
        public String getName(  );
        public double getVersion(  );
        public Set keySet(  );
        public synchronized void load(InputStream);
        public synchronized Object put(Object, Object);
        public synchronized void putAll(Map);
        public synchronized Object remove(Object);
        public String toString(  );
        public Collection values(  );
}
```

See also

`Security`

Interface java.security.PublicKey

A public key is a key with certain mathematical properties that allows it to perform inverse cryptographic operations with its matching private key. Classes implement this interface only for type identification.

Interface Definition

```
public interface java.security.PublicKey
        implements java.security.Key {
}
```

See also

`Key`, *`PrivateKey`*

Class java.security.SecureClassLoader

A secure class loader is a class loader that is able to associate code sources (and hence protection domains) with the classes that it loads (classes loaded by a traditional class loader have a default, null protection domain). All new class loaders are expected to extend this class.

Class Definition

```
public class java.security.SecureClassLoader
        extends java.lang.ClassLoader {

        // Constructors
        protected SecureClassLoader(  );
        protected SecureClassLoader(ClassLoader);

        // Protected Instance Methods
        protected final Class defineClass(String, byte[], int,
                                          int, CodeSource);
        protected PermissionCollection getPermissions(CodeSource);
}
```

See also

`ClassLoader`, *`CodeSource`*, *`ProtectionDomain`*, *`java.net.URLClassLoader`*

Class java.security.SecureRandom

This class generates random numbers. Unlike the standard random−number generator, numbers generated by this class are cryptographically secure −− that is, they are less subject to pattern guessing and other attacks that can be made upon a traditional random−number generator.

Class Definition

```
public class java.security.SecureRandom
    extends java.util.Random {

    // Constructors
    public SecureRandom(byte[]);
    protected SecureRandom(SecureRandomSpi, Provider);
    public SecureRandom(  );

    // Class Methods
    public static SecureRandom getInstance(String, String);
    public static SecureRandom getInstance(String);
    public static byte[] getSeed(int);

    // Instance Methods
    public byte[] generateSeed(int);
    public final Provider getProvider(  );
    public synchronized void nextBytes(byte[]);
    public synchronized void setSeed(byte[]);
    public void setSeed(long);
    // Protected Instance Methods
    protected final int next(int);
}
```

Class java.security.SecureRandomSpi

This is the service provider interface for the secure random number generator. Instances of this engine can be registered with a security provider and used to generate the random numbers required by the security algorithms.

Class Definition

```
public abstract class java.security.SecureRandomSpi
    extends java.lang.Object
    implements java.io.Serializable {

    // Constructors
    public SecureRandomSpi(  );

    // Protected Instance Methods
    protected abstract byte[] engineGenerateSeed(int);
    protected abstract void engineNextBytes(byte[]);
    protected abstract void engineSetSeed(byte[]);
}
```

Class java.security.Security

This class manages the list of providers that have been installed into the virtual machine; this list of providers is consulted to find an appropriate class to provide the implementation of a particular operation when the `getInstance( )` method of an engine class is called. The list of providers initially comes from the *$JDKHOME/lib/security/java.security* file, and applications may use methods of this class to add and remove providers from that list.

Class Definition

```
public final java.security.Security
    extends java.lang.Object {

    // Class Methods
    public static int addProvider(Provider);
    public static String getAlgorithmProperty(String, String);
    public static String getProperty(String);
    public static synchronized Provider getProvider(String);
    public static Provider[] getProviders(String);
    public static synchronized Provider[] getProviders(  );
    public static Provider[] getProviders(Map);
    public static synchronized int insertProviderAt(Provider, int);
    public static synchronized void removeProvider(String);
    public static void setProperty(String, String);
}
```

See also

Provider

Class java.security.SecurityPermission

This class represents permissions to interact with the methods of the `java.security` package. This permission is a basic permission; it does not support actions. Security permissions are checked by the `Identity`, `Signer`, and `Provider` classes.

Class Definition

```
public final class java.security.SecurityPermission
        extends java.security.BasicPermission {

        // Constructors
        public SecurityPermission(String);
        public SecurityPermission(String, String);
}
```

See also

BasicPermission

Class java.security.Signature

This engine class provides the ability to create or verify digital signatures by employing different algorithms that have been registered with the `Security` class. As with all engine classes, instances of this class are obtained via the `getInstance( )` method. The signature object must be initialized with the appropriate private key (to sign) or public key (to verify), then data must be fed to the object via the `update( )` methods, and then the signature can be obtained (via the `sign( )` method) or verified (via the `verify( )` method). Signature objects may support algorithm−specific parameters, though this is not a common implementation.

Class Definition

```
public abstract class java.security.Signature
        extends java.security.SignatureSpi {

        // Constants
        protected static final int SIGN;
        protected static final int UNINITIALIZED;
        protected static final int VERIFY;

        // Variables
        protected int state;

        // Constructors
        protected Signature(String);
        // Class Methods
        public static Signature getInstance(String);
        public static Signature getInstance(String, String);

        // Instance Methods
        public Object clone(  );
        public final String getAlgorithm(  );
        public final Object getParameter(String);
        public final Provider getProvider(  );
        public final void initSign(PrivateKey);
        public final void initSign(PrivateKey, SecureRandom);
        public final void initVerify(PublicKey);
        public final void initVerify(Certificate);
        public final void setParameter(String, Object);
        public final void setParameter(AlgorithmParameterSpec);
        public final byte[] sign(  );
        public final int sign(byte[], int, int);
        public String toString(  );
        public final void update(byte);
        public final void update(byte[]);
        public final void update(byte[], int, int);
        public final boolean verify(byte[]);
}
```

See also

`Provider`

Class java.security.SignatureSpi

This is the service provider interface for the signature engine. If you want to implement your own signature engine, you must extend this class and register your implementation with an appropriate security provider. Since the `Signature` class already extends this class, your implementation may extend the `Signature` class directly. Implementations of this class must be prepared both to sign and to verify data that is passed to the `engineUpdate( )` method. Initialization of the engine may optionally support a set of algorithm−specific parameters.

Class Definition

```
public abstract class java.security.SignatureSpi
        extends java.lang.Object {

        // Variables
        protected SecureRandom appRandom;

        // Constructors
        public SignatureSpi(  );

        // Instance Methods
        public Object clone(  );

        // Protected Instance Methods
        protected abstract Object engineGetParameter(String);
        protected abstract void engineInitSign(PrivateKey);
        protected void engineInitSign(PrivateKey, SecureRandom);
        protected abstract void engineInitVerify(PublicKey);
        protected abstract void engineSetParameter(String, Object);
        protected void engineSetParameter(AlgorithmParameterSpec);
        protected abstract byte[] engineSign(  );
        protected final int engineSign(byte[], int, int);
        protected abstract void engineUpdate(byte);
        protected abstract void engineUpdate(byte[], int, int);
        protected abstract boolean engineVerify(byte[]);
}
```

See also

Provider, *Signature*

Class java.security.SignedObject

A signed object is a container class for another (target) object; the signed object contains a serialized version of the target along with a digital signature of the data contained in the target object. You must provide a serializable object and a private key to create a signed object, after which you can remove the embedded object and verify the signature of the signed object by providing the appropriate public key.

Class Definition

```
public final class java.security.SignedObject
        extends java.lang.Object
        implements java.io.Serializable {

        // Constructors
        public SignedObject(Serializable, PrivateKey, Signature);

        // Instance Methods
        public String getAlgorithm(  );
        public Object getObject(  );
        public byte[] getSignature(  );
        public boolean verify(PublicKey, Signature);
}
```

See also

Signature

Class java.security.Signer

A signer abstracts the notion of a principal (that is, an individual or a corporation) that has a private key and a corresponding public key. Signers may optionally belong to an identity scope. This class is deprecated.

Class Definition

```
public abstract class java.security.Signer
        extends java.security.Identity {

        // Constructors
        protected Signer(  );
        public Signer(String);
        public Signer(String, IdentityScope);

        // Instance Methods
        public PrivateKey getPrivateKey(  );
        public final void setKeyPair(KeyPair);
        public String toString(  );
}
```

See also

`Identity`, *`Principal`*

Class java.security.UnresolvedPermission

An unresolved permission is one for which the implementing class has not been loaded. If you define a custom permission, the `Policy` class will represent that custom permission as an unresolved permission until it is time for the `Policy` class to actually load the class; if the class cannot be found, then it will remain an unresolved permission. By default, the `implies( )` method of this class always returns `false`.

Class Definition

```
public final class java.security.UnresolvedPermission
    extends java.security.Permission
    implements java.io.Serializable {

    // Constructors
    public UnresolvedPermission(String, String, String, Certificate[]);

    // Instance Methods
    public boolean equals(Object);
    public String getActions(  );
    public int hashCode(  );
    public boolean implies(Permission);
    public PermissionCollection newPermissionCollection(  );
    public String toString(  );
}
```

See also

`Permission`

F.2 Package java.security.cert

Class java.security.cert.Certificate

This class represents any type of cryptographic certificate. A certificate contains a public key (see `getPublicKey( )`) and other associated information. The certificate contains an internal signature that protects its integrity. You can verify the integrity of the certificate by calling one of the `verify( )` methods with the public key of the certificate's issuer. (Note: don't confuse this class with the `java.security.Certificate` interface, which is deprecated.)

Class Definition

```
public abstract class java.security.cert.Certificate
    extends java.lang.Object
    implements java.io.Serializable {

    // Constructors
    protected Certificate(String);

    // Instance Methods
    public boolean equals(Object);
    public abstract byte[] getEncoded(  );
    public abstract PublicKey getPublicKey(  );
    public final String getType(  );
    public int hashCode(  );
    public abstract String toString(  );
    public abstract void verify(PublicKey);
    public abstract void verify(PublicKey, String);

    // Protected Instance Methods
    protected Object writeReplace(  );
}
```

See also

`PublicKey`, *`X509Certificate`*

Class java.security.cert.CertificateFactory

A certificate factory is used to import certificates or certificate revocation lists from a file or other input stream.

Class Definition

```
public class java.security.cert.CertificateFactory
        extends java.lang.Object{

        //Constructors
        protected CertificateFactory(CertificateFactorySpi, Provider,
                                                                String);
        //Class Methods
        public static final CertificateFactory getInstance(String);
        public static final CertificateFactory getInstance(String,
                                                                 String);
        //Instance Methods
        public final CRL generateCRL(InputStream);
        public final Collection generateCRLs(InputStream);
        public final Certificate generateCertificate(InputStream);
        public final Collection generateCertificates(InputStream);
        public final Provider getProvider(  );
        public final String getType(  );
}
```

See also

X509Certificate, *X509CRLEntry*

Class java.security.cert.CertificateFactorySpi

This is the service provider interface used to define a certificate factory. It is responsible for decoding certificates and certificate revocation lists read from the given input stream.

Class Definition

```
public abstract class java.security.cert.CertificateFactorySpi
    extends java.lang.Object {

    // Constructors
    public CertificateFactorySpi(  );

    // Instance Methods
    public abstract CRL engineGenerateCRL(InputStream);
    public abstract Collection engineGenerateCRLs(InputStream);
    public abstract Certificate engineGenerateCertificate(InputStream);
    public abstract Collection engineGenerateCertificates(InputStream);
}
```

Class java.security.cert.CRL

This class represents a generic certificate revocation list (CRL). Instances of this class are obtained from a certificate factory, and you can use those instances to determine if a particular certificate has been revoked.

Class Definition

```
public abstract class java.security.cert.CRL
    extends java.lang.Object {

    // Constructors
    protected CRL(String);

    // Instance Methods
    public final String getType(  );
    public abstract boolean isRevoked(Certificate);
    public abstract String toString(  );
}
```

See also

X509CRL

Class java.security.cert.X509Certificate

This class represents certificates as defined in the X.509 standard. Such certificates associate a public key with a subject, which is usually a person or organization. You can find out the certificate's subject by calling `getSubjectDN( )` while you can retrieve the subject's public key using `getPublicKey( )`. The certificate's issuer is the person or organization that generated and signed the certificate (see `getIssuerDN( )`). If you have a certificate file in the format described by RFC 1421, you can create an X509Certificate from that data by using one of the `getInstance( )` methods.

Class Definition

```
public abstract class java.security.cert.X509Certificate
    extends java.security.cert.Certificate
    implements java.security.cert.X509Extension {

    // Constructors
    protected X509Certificate(  );

    // Instance Methods
    public abstract void checkValidity(  );
    public abstract void checkValidity(Date);
    public abstract int getBasicConstraints(  );
    public abstract Set getCriticalExtensionOIDs(  );
    public abstract byte[] getExtensionValue(String);
    public abstract Principal getIssuerDN(  );
    public abstract boolean[] getIssuerUniqueID(  );
    public abstract boolean[] getKeyUsage(  );
    public abstract Set getNonCriticalExtensionOIDs(  );
    public abstract Date getNotAfter(  );
    public abstract Date getNotBefore(  );
    public abstract BigInteger getSerialNumber(  );
    public abstract String getSigAlgName(  );
    public abstract String getSigAlgOID(  );
    public abstract byte[] getSigAlgParams(  );
    public abstract byte[] getSignature(  );
    public abstract Principal getSubjectDN(  );
    public abstract boolean[] getSubjectUniqueID(  );
    public abstract byte[] getTBSCertificate(  );
    public abstract int getVersion(  );
    public abstract boolean hasUnsupportedCriticalExtension(  );
}
```

See also

`Principal`, *`PublicKey`*, *`X509Extension`*

Class java.security.cert.X509CRL

A Certificate Revocation List (CRL) is a list of certificates whose keys are no longer valid. This class represents CRLs as defined in the X.509 standard. If you have a CRL file that you would like to examine, you can construct an `X509CRL` object from the file using one of the `getInstance( )` methods. A CRL, just like a certificate, has an internal signature that protects its integrity. To verify the integrity of the CRL itself, call one of the `verify( )` methods with the issuer's public key. To find out if a particular certificate is revoked, call the `isRevoked( )` method with the certificate's serial number.

Class Definition

```
public abstract class java.security.cert.X509CRL
    extends java.security.cert.CRL
    implements java.security.cert.X509Extension {

    // Constructors
    protected X509CRL(  );

    // Instance Methods
    public boolean equals(Object);
    public abstract Set getCriticalExtensionOIDs(  );
    public abstract byte[] getEncoded(  );
    public abstract byte[] getExtensionValue(String);
    public abstract Principal getIssuerDN(  );
    public abstract Date getNextUpdate(  );
    public abstract Set getNonCriticalExtensionOIDs(  );
    public abstract X509CRLEntry getRevokedCertificate(BigInteger);
    public abstract Set getRevokedCertificates(  );
    public abstract String getSigAlgName(  );
    public abstract String getSigAlgOID(  );
    public abstract byte[] getSigAlgParams(  );
    public abstract byte[] getSignature(  );
    public abstract byte[] getTBSCertList(  );
    public abstract Date getThisUpdate(  );
    public abstract int getVersion(  );
    public abstract boolean hasUnsupportedCriticalExtension(  );
    public int hashCode(  );
    public abstract void verify(PublicKey, String);
    public abstract void verify(PublicKey);
}
```

See also

Certificate, *PublicKey*, *X509CRLEntry*, *X509Extension*

Class java.security.cert.X509CRLEntry

A revoked certificate represents a certificate whose contained key is no longer safe to use. Instances of this class are returned by X509CRL's `getRevoked−Certificate( )` method. You can examine the certificate's revocation date and X.509 extensions.

Class Definition

```
public abstract class java.security.cert.X509CRLEntry
    extends java.lang.Object
    implements java.security.cert.X509Extension {

    // Constructors
    public X509CRLEntry(  );

    // Instance Methods
    public boolean equals(Object);
    public abstract Set getCriticalExtensionOIDs(  );
    public abstract byte[] getEncoded(  );
    public abstract byte[] getExtensionValue(String);
    public abstract Set getNonCriticalExtensionOIDs(  );
    public abstract Date getRevocationDate(  );
    public abstract BigInteger getSerialNumber(  );
    public abstract boolean hasExtensions(  );
    public abstract boolean hasUnsupportedCriticalExtension(  );
    public int hashCode(  );
    public abstract String toString(  );
}
```

See also

`Certificate`, *`X509CRL`*, *`X509Extension`*

Interface java.security.cert.X509Extension

The X509Extension interface represents the certificate extensions defined by the X.509v3 standard. Extensions are additional bits of information contained in a certificate. Each extension is designated as critical or noncritical. An application that handles a certificate should either correctly interpret the critical extensions or produce some kind of error if they cannot be recognized.

Class Definition

```
public interface java.security.cert.X509Extension {

        // Instance Methods
        public abstract Set getCriticalExtensionOIDs(  );
        public abstract boolean hasUnsupportedCriticalExtension(  );
        public abstract byte[] getExtensionValue(String);
        public abstract Set getNonCriticalExtensionOIDs(  );
}
```

See also

`X509CRLEntry`, *`X509Certificate`*, *`X509CRL`*

F.3 Package java.security.interfaces

Interface java.security.interfaces.DSAKey

This interface represents public and private keys that are suitable for use in DSA signature algorithms. It allows you to retrieve DSA−specific information from a suitable DSA key.

Interface Definition

```
public interface java.security.interfaces.DSAKey {

        // Instance Methods
        public DSAParams getParams(  );
}
```

See also

PrivateKey, *PublicKey*

Interface java.security.interfaces.DSAKeyPair−Generator

This interface represents key generators that can be used to generate pairs of DSA keys. Key pair generators that implement this interface can be initialized with information specific to DSA key generation.

Interface Definition

```
public interface java.security.interfaces.DSAKeyPairGenerator {

        // Instance Methods
        public void initialize(DSAParams, SecureRandom);
        public void initialize(int, boolean, SecureRandom);
}
```

See also

KeyPairGenerator

Interface java.security.interfaces.DSAParams

Classes that implement this interface allow you to obtain the three variables that are common to both DSA public and private keys.

Interface Definition

```
public interface java.security.interfaces.DSAParams {

        // Instance Methods
        public BigInteger getP(  );
        public BigInteger getQ(  );
        public BigInteger getG(  );
}
```

See also

DSAPrivateKey, *DSAPublicKey*

Interface java.security.interfaces.DSAPrivateKey

Classes that implement this interface allow you to retrieve the private key parameter used to calculate a DSA private key.

Interface Definition

```
public interface java.security.interfaces.DSAPrivateKey {

        // Instance Methods
        public BigInteger getX(  );
}
```

See also

DSAParams, *DSAPublicKey*

Interface java.security.interfaces.DSAPublicKey

Classes that implement this interface allow you to retrieve the public key parameter used to calculate a DSA public key.

Interface Definition

```
public interface java.security.interfaces.DSAPublicKey {

        // Instance Methods
        public BigInteger getY(  );
}
```

See also

DSAParams, *DSAPrivateKey*

Interface java.security.interfaces.RSAKey

Classes that implement this interface represent an RSA public or private key.

Interface Definition

```
public interface java.security.interfaces.RSAKey {

    // Instance Methods
    public abstract BigInteger getModulus(  );
}
```

Interface java.security.interfaces.RSAPrivateCrtKey

A class that implements this interface represents an RSA private key that uses the Chinese Remainder Theorem to calculate its value.

Class Definition

```
public interface java.security.interfaces.RSAPrivateCrtKey
    implements java.security.interfaces.RSAPrivateKey {

    // Instance Methods
    public abstract BigInteger getCrtCoefficient(  );
    public abstract BigInteger getPrimeExponentP(  );
    public abstract BigInteger getPrimeExponentQ(  );
    public abstract BigInteger getPrimeP(  );
    public abstract BigInteger getPrimeQ(  );
    public abstract BigInteger getPublicExponent(  );
}
```

Interface java.security.interfaces.RSAPrivateKey

This class represents a private key that is suitable for use with RSA cryptographic operations.

Interface Definition

```
public interface java.security.interfaces.RSAPrivateKey
        implements java.security.PrivateKey {

        // Instance Methods
        public abstract BigInteger getPrivateExponent(  );
}
```

See also

PrivateKey, *RSAPublicKey*

Interface java.security.interfaces.RSAPublicKey

This class represents an RSA public key, suitable for use with an RSA cryptographic algorithm.

Interface Definition

```
public interface java.security.interfaces.RSAPublicKey
        implements java.security.PublicKey {

        // Instance Methods
        public abstract BigInteger getPublicExponent(  );
}
```

See also

PublicKey, *RSAPrivateKey*

F.4 Package java.security.spec

Interface java.security.spec.Algorithm−ParameterSpec

Algorithm parameter specifications are used to import and export keys via a key factory. This interface is used strictly for type identification; the specifics of the parameters are left to the implementing class.

Interface Definition

```
public interface java.security.spec.AlgorithmParameterSpec {
}
```

See also

DSAParameterSpec, *KeyFactory*

Class java.security.spec.DSAParameterSpec

This class provides the basis for DSA key generation via parameters; it encapsulates the three parameters that are common to DSA algorithms.

Class Definition

```
public class java.security.spec.DSAParameterSpec
        extends java.lang.Object
        implements java.security.spec.AlgorithmParameterSpec,
                         java.security.interfaces.DSAParams {

        // Constructors
        public DSAParameterSpec(BigInteger, BigInteger, BigInteger);

        // Instance Methods
        public BigInteger getG(  );
        public BigInteger getP(  );
        public BigInteger getQ(  );
}
```

See also

AlgorithmParameterSpec, *DSAParams*, *DSAPrivateKeySpec*, *DSAPublicKeySpec*

Class java.security.spec.DSAPrivateKeySpec

This class provides the ability to calculate a DSA private key based upon the four parameters that comprise the key.

Class Definition

```
public class java.security.spec.DSAPrivateKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        // Constructors
        public DSAPrivateKeySpec(BigInteger, BigInteger,
                                      BigInteger, BigInteger);

        // Instance Methods
        public BigInteger getG(  );
        public BigInteger getP(  );
        public BigInteger getQ(  );
        public BigInteger getX(  );
}
```

See also

DSAPublicKeySpec, *KeyFactory*

Class java.security.spec.DSAPublicKeySpec

This class provides the ability to calculate a DSA public key based upon the four parameters that comprise the key.

Class Definition

```
public class java.security.spec.DSAPublicKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        // Constructors
        public DSAPublicKeySpec(BigInteger, BigInteger,
                                     BigInteger, BigInteger);

        // Instance Methods
        public BigInteger getG(  );
        public BigInteger getP(  );
        public BigInteger getQ(  );
        public BigInteger getY(  );
}
```

See also

DSAPrivateKeySpec, *KeyFactory*

Class java.security.spec.EncodedKeySpec

This class is used to translate between keys and their external encoded format. The encoded format is always simply a series of bytes, but the format of the encoding of the key information into those bytes may vary depending upon the algorithm used to generate the key.

Class Definition

```
public abstract class java.security.spec.EncodedKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        // Constructors
        public EncodedKeySpec(  );

        // Instance Methods
        public abstract byte[] getEncoded(  );
        public abstract String getFormat(  );
}
```

See also

KeyFactory, *KeySpec*, *PKCS8EncodedKeySpec*, *X509EncodedKeySpec*

Interface java.security.spec.KeySpec

A key specification is used to import and export keys via a key factory. This may be done either based upon the algorithm parameters used to generate the key or via an encoded series of bytes that represent the key. Classes that deal with the latter case implement this interface, which is used strictly for type identification.

Interface Definition

```
public interface java.security.spec.KeySpec {
}
```

See also

AlgorithmParameterSpec, *EncodedKeySpec*, *KeyFactory*

Class java.security.spec.PKCS8EncodedKeySpec

This class represents the PKCS#8 encoding of a private key; the key is encoded in DER format. This is the class that is typically used when dealing with DSA private keys in a key factory.

Class Definition

```
public class java.security.spec.PKCS8EncodedKeySpec
        extends java.security.spec.EncodedKeySpec {

        // Constructors
        public PKCS8EncodedKeySpec(byte[]);

        // Instance Methods
        public byte[] getEncoded(  );
        public final String getFormat(  );
}
```

See also

`EncodedKeySpec`, *`X509EncodedKeySpec`*

Class java.security.spec.RSAKeyGenParameterSpec

This class encapsulates the RSA key generation parameters for use with a key factory.

Class Definition

```
public class java.security.spec.RSAKeyGenParameterSpec
    extends java.lang.Object
    implements java.security.spec.AlgorithmParameterSpec {

    // Constants
    public static final BigInteger F0;
    public static final BigInteger F4;

    // Constructors
    public RSAKeyGenParameterSpec(int, BigInteger);

    // Instance Methods
    public int getKeysize(  );
    public BigInteger getPublicExponent(  );
}
```

Class java.security.spec.RSAPrivateCrtKeySpec

This class represents the algorithm parameters for an RSA private key that uses the Chinese Remainder Theorem to calculate its value.

Class Definition

```
public class java.security.spec.RSAPrivateCrtKeySpec
    extends java.security.spec.RSAPrivateKeySpec {

    // Constructors
    public RSAPrivateCrtKeySpec(BigInteger, BigInteger, BigInteger,
                     BigInteger, BigInteger, BigInteger,
                     BigInteger, BigInteger);

    // Instance Methods
    public BigInteger getCrtCoefficient(  );
    public BigInteger getPrimeExponentP(  );
    public BigInteger getPrimeExponentQ(  );
    public BigInteger getPrimeP(  );
    public BigInteger getPrimeQ(  );
    public BigInteger getPublicExponent(  );
}
```

Class java.security.spec.RSAPrivateKeySpec

This class represents a key specification for an RSA private key; this specification uses a modulus and a private exponent. Instances of this class may be used with an appropriate key factory to generate private keys.

Class Definition

```
public class java.security.spec.RSAPrivateKeySpec
    extends java.lang.Object
    implements java.security.spec.KeySpec {

    // Constructors
    public RSAPrivateKeySpec(BigInteger, BigInteger);

    // Instance Methods
    public BigInteger getModulus(  );
    public BigInteger getPrivateExponent(  );
}
```

See also

KeyFactory, *KeySpec*, *PrivateKey*

Class java.security.spec.RSAPublicKeySpec

This class represents a key specification for an RSA public key. Instances of this class may be used with an appropriate key factory to generate public keys.

Class Definition

```
public class java.security.spec.RSAPublicKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        // Constructors
        public RSAPublicKeySpec(BigInteger, BigInteger);

        // Instance Methods
        public BigInteger getModulus(  );
        public BigInteger getPublicExponent(  );
}
```

See also

KeyFactory, *KeySpec*, *PublicKey*

Class java.security.spec.X509EncodedKeySpec

This class represents the X509 encoding of a public key. It may also be used for private keys, although the PKCS#8 encoding is typically used for those keys.

Class Definition

```
public class java.security.spec.X509EncodedKeySpec
        extends java.security.spec.EncodedKeySpec {

        // Constructors
        public X509EncodedKeySpec(byte[]);

        // Instance Methods
        public byte[] getEncoded(  );
        public final String getFormat(  );
}
```

See also

`EncodedKeySpec`, *`PKCS8EncodedKeySpec`*

F.5 Package javax.crypto

Class javax.crypto.Cipher

This engine class represents a cryptographic cipher, either symmetric or asymmetric. To get a cipher for a particular algorithm, call one of the `getInstance( )` methods, specifying an algorithm name, a cipher mode, and a padding scheme. The cipher should be initialized for encryption or decryption using an `init( )` method and an appropriate key (and, optionally, a set of algorithm−specific parameters, though these are typically unused). Then you can perform the encryption or decryption using the `update( )` and `doFinal( )` methods.

Class Definition

```
public class javax.crypto.Cipher
    extends java.lang.Object {

    // Constants
    public static final int DECRYPT_MODE;
    public static final int ENCRYPT_MODE;
    public static final int PRIVATE_KEY;
    public static final int PUBLIC_KEY;
    public static final int SECRET_KEY;
    public static final int UNWRAP_MODE;
    public static final int WRAP_MODE;

    // Constructors
    protected Cipher(CipherSpi, Provider, String);

    // Class Methods
    public static final Cipher getInstance(String);
    public static final Cipher getInstance(String, String);

    // Instance Methods
    public final int doFinal(byte[], int, int, byte[]);
    public final int doFinal(byte[], int, int, byte[], int);
    public final byte[] doFinal(byte[]);
    public final byte[] doFinal(byte[], int, int);
    public final byte[] doFinal(  );
    public final int doFinal(byte[], int);
    public final String getAlgorithm(  );
    public final int getBlockSize(  );
    public final ExemptionMechanism getExemptionMechanism(  );
    public final byte[] getIV(  );
    public final int getOutputSize(int);
    public final AlgorithmParameters getParameters(  );
    public final Provider getProvider(  );
    public final void init(int, Key, AlgorithmParameterSpec);
    public final void init(int, Key);
    public final void init(int, Certificate, SecureRandom);
    public final void init(int, Certificate);
    public final void init(int, Key,
                     AlgorithmParameterSpec, SecureRandom);
    public final void init(int, Key,
                     AlgorithmParameters, SecureRandom);
    public final void init(int, Key, SecureRandom);
    public final void init(int, Key, AlgorithmParameters);
    public final Key unwrap(byte[], String, int);
    public final byte[] update(byte[]);
    public final int update(byte[], int, int, byte[]);
    public final byte[] update(byte[], int, int);
    public final int update(byte[], int, int, byte[], int);
    public final byte[] wrap(Key);
}
```

See also

AlgorithmParameterSpec, *CipherSpi*, *Key*, *Provider*, *SecureRandom*

Class javax.crypto.CipherInputStream

A cipher input stream is a filter stream that passes its data through a cipher. You can construct a cipher input stream by specifying an underlying stream and supplying an initialized cipher. For best results, use a byte−oriented cipher mode with this stream.

Class Definition

```
public class javax.crypto.CipherInputStream
        extends java.io.FilterInputStream {

        // Constructors
        protected CipherInputStream(InputStream);
        public CipherInputStream(InputStream, Cipher);

        // Instance Methods
        public int available(  );
        public void close(  );
        public boolean markSupported(  );
        public int read(  );
        public int read(byte[]);
        public int read(byte[], int, int);
        public long skip(long);
}
```

See also

Cipher

Class javax.crypto.CipherOutputStream

This class is a filter output stream that passes all its data through a cipher. You can construct a cipher output stream by specifying an underlying output stream and an initialized cipher. For best results, use a byte−oriented mode for the cipher.

Class Definition

```
public class javax.crypto.CipherOutputStream
        extends java.io.FilterOutputStream {

        // Constructors
        protected CipherOutputStream(OutputStream);
        public CipherOutputStream(OutputStream, Cipher);

        // Instance Methods
        public void close(  );
        public void flush(  );
        public void write(int);
        public void write(byte[]);
        public void write(byte[], int, int);
}
```

See also

Cipher

Class javax.crypto.CipherSpi

This class is the service provider interface of the `Cipher` class. To implement a particular cipher algorithm, create a subclass of this class and register the class with an appropriate security provider. Like all SPI classes, the methods that begin with engine are called by their corresponding method (without engine) from the `Cipher` class. This is a JCE engine, which means it must be deployed in a specially signed jar file.

Class Definition

```
public abstract class javax.crypto.CipherSpi
    extends java.lang.Object {

    // Constructors
    public CipherSpi(  );

    // Protected Instance Methods
    protected abstract int engineDoFinal(byte[], int, int,
                            byte[], int);
    protected abstract byte[] engineDoFinal(byte[], int, int);
    protected abstract int engineGetBlockSize(  );
    protected abstract byte[] engineGetIV(  );
    protected int engineGetKeySize(Key);
    protected abstract int engineGetOutputSize(int);
    protected abstract AlgorithmParameters engineGetParameters(  );
    protected abstract void engineInit(int, Key,
                            AlgorithmParameters, SecureRandom);
    protected abstract void engineInit(int, Key, SecureRandom);
    protected abstract void engineInit(int, Key,
                            AlgorithmParameterSpec, SecureRandom);
    protected abstract void engineSetMode(String);
    protected abstract void engineSetPadding(String);
    protected Key engineUnwrap(byte[], String, int);
    protected abstract int engineUpdate(byte[], int, int, byte[], int);
    protected abstract byte[] engineUpdate(byte[], int, int);
    protected byte[] engineWrap(Key);
}
```

See also

`AlgorithmParameterSpec`, *`Cipher`*, *`Key`*, *`SecureRandom`*

Class javax.crypto.ExemptionMechanism

This class can be used by implementors of a JCE package to provide hooks into the exemption mechanisms of various governments. This is done in order to satisfy the import and/or export restrictions of those governments, who might require (for example) a key escrow implemented with this class. It is not used by Sun's implementation of JCE.

Class Definition

```
public class javax.crypto.ExemptionMechanism
    extends java.lang.Object {

    // Constructors
    protected ExemptionMechanism(ExemptionMechanismSpi,
                                 Provider, String);

    // Class Methods
    public static final ExemptionMechanism getInstance(String);
    public static final ExemptionMechanism getInstance(String, String);

    // Instance Methods
    public final int genExemptionBlob(byte[], int);
    public final byte[] genExemptionBlob(  );
    public final int genExemptionBlob(byte[]);
    public final String getName(  );
    public final int getOutputSize(int);
    public final Provider getProvider(  );
    public final void init(Key, AlgorithmParameterSpec);
    public final void init(Key, AlgorithmParameters);
    public final void init(Key);
    public final boolean isCryptoAllowed(Key);

    // Protected Instance Methods
    protected void finalize(  );
}
```

Class javax.crypto.ExemptionMechanismSpi

This is the service provider interface for developers of providers that must include special exemption algorithms, such as key escrow. This class is unused by Sun's implementation of JCE. Since it is a JCE engine, it must be deployed in a specially signed jar file.

Class Definition

```
public abstract class javax.crypto.ExemptionMechanismSpi
    extends java.lang.Object {

    // Constructors
    public ExemptionMechanismSpi(  );

    // Protected Instance Methods
    protected abstract int engineGenExemptionBlob(byte[], int);
    protected abstract byte[] engineGenExemptionBlob(  );
    protected abstract int engineGetOutputSize(int);
    protected abstract void engineInit(Key, AlgorithmParameters);
    protected abstract void engineInit(Key);
    protected abstract void engineInit(Key, AlgorithmParameterSpec);
}
```

Class javax.crypto.KeyAgreement

This engine class represents a key agreement protocol, which is an arrangement by which two parties can agree on a secret value. You can obtain an instance of this class by calling the `getInstance( )` method. After initializing the object (see `init( )`), you can step through the phases of the key agreement protocol using the `doPhase( )` method. Once the phases are complete, the secret value (that is, the key) is returned from the `generateSecret( )` method.

Class Definition

```
public class javax.crypto.KeyAgreement
        extends java.lang.Object {

        // Constructors
        protected KeyAgreement(KeyAgreementSpi, Provider, String);

        // Class Methods
        public static final KeyAgreement getInstance(String);
        public static final KeyAgreement getInstance(String, String);

        // Instance Methods
        public final Key doPhase(Key, boolean);
        public final byte[] generateSecret(  );
        public final int generateSecret(byte[], int);
        public final SecretKey generateSecret(String);
        public final String getAlgorithm(  );
        public final Provider getProvider(  );
        public final void init(Key);
        public final void init(Key, SecureRandom);
        public final void init(Key, AlgorithmParameterSpec);
        public final void init(Key, AlgorithmParameterSpec, SecureRandom);
}
```

See also

`AlgorithmParameterSpec`, *`Key`*, *`KeyAgreementSpi`*, *`Provider`*, *`SecureRandom`*

Class javax.crypto.KeyAgreementSpi

This is the service provider interface class for the `KeyAgreement` class. If you want to implement a key agreement algorithm, create a subclass of this class and register it with an appropriate security provider. Because it is a JCE engine, implementations of this class must be deployed in a specially signed jar file.

Class Definition

```
public abstract class javax.crypto.KeyAgreementSpi
        extends java.lang.Object {

        // Constructors
        public KeyAgreementSpi(  );

        // Protected Instance Methods
        protected abstract Key engineDoPhase(Key, boolean);
        protected abstract byte[] engineGenerateSecret(  );
        protected abstract int engineGenerateSecret(byte[], int);
        protected abstract SecretKey engineGenerateSecret(String);
        protected abstract void engineInit(Key, SecureRandom);
        protected abstract void engineInit(Key, AlgorithmParameterSpec,
                                           SecureRandom);
}
```

See also

`AlgorithmParameterSpec`, *`Key`*, *`KeyAgreement`*, *`SecureRandom`*

Class javax.crypto.KeyGenerator

A key generator creates secret keys for use with symmetric ciphers. Key generators are obtained by calling the `getInstance( )` method; they must then be initialized with an `init( )` method. The key itself is then returned from the `generateSecret( )` method.

Class Definition

```
public class javax.crypto.KeyGenerator
        extends java.lang.Object {

        // Constructors
        protected KeyGenerator(KeyGeneratorSpi, Provider, String);

        // Class Methods
        public static final KeyGenerator getInstance(String);
        public static final KeyGenerator getInstance(String, String);
        // Instance Methods
        public final SecretKey generateKey(  );
        public final String getAlgorithm(  );
        public final Provider getProvider(  );
        public final void init(int);
        public final void init(int, SecureRandom);
        public final void init(SecureRandom);
        public final void init(AlgorithmParameterSpec);
        public final void init(AlgorithmParameterSpec, SecureRandom);
}
```

See also

`AlgorithmParameterSpec`, *`KeyGeneratorSpi`*, *`Provider`*, *`SecretKey`*, *`SecureRandom`*

Class javax.crypto.KeyGeneratorSpi

This is the service provider interface for the `KeyGenerator` class. To create an implementation of a key generation algorithm, make a subclass of this class and register the implementation with an appropriate security provider. Because it is a JCE engine, it must be deployed in a specially signed jar file.

Class Definition

```
public abstract class javax.crypto.KeyGeneratorSpi
        extends java.lang.Object {

        // Constructors
        public KeyGeneratorSpi(  );

        // Protected Instance Methods
        protected abstract SecretKey engineGenerateKey(  );
        protected abstract void engineInit(int, SecureRandom);
        protected abstract void engineInit(SecureRandom);
        protected abstract void engineInit(AlgorithmParameterSpec,
                                           SecureRandom);
}
```

See also

`AlgorithmParameterSpec`, *`KeyGenerator`*, *`SecretKey`*, *`SecureRandom`*

Class javax.crypto.Mac

This is a secure message digest, otherwise known as a Message Authentication Code. This class uses a secret key to perform additional calculations on a message digest (resulting in a MAC), making it impossible to change the original data without the secret key.

Class Definition

```
public class javax.crypto.Mac
    extends java.lang.Object
    implements java.lang.Cloneable {

    // Constructors
    protected Mac(MacSpi, Provider, String);

    // Class Methods
    public static final Mac getInstance(String, String);
    public static final Mac getInstance(String);

    // Instance Methods
    public final Object clone(  );
    public final byte[] doFinal(byte[]);
    public final byte[] doFinal(  );
    public final void doFinal(byte[], int);
    public final String getAlgorithm(  );
    public final int getMacLength(  );
    public final Provider getProvider(  );
    public final void init(Key, AlgorithmParameterSpec);
    public final void init(Key);
    public final void reset(  );
    public final void update(byte);
    public final void update(byte[], int, int);
    public final void update(byte[]);
}
```

See also

java.security.MessageDigest

Class javax.crypto.MacSpi

This is the service provider interface for developers who want to provide their own MAC algorithms. Because it is a JCE engine, instances of this class must be deployed in a specially−signed jar file.

Class Definition

```
public abstract class javax.crypto.MacSpi
    extends java.lang.Object {

    // Constructors
    public MacSpi(  );

    // Instance Methods
    public Object clone(  );

    // Protected Instance Methods
    protected abstract byte[] engineDoFinal(  );
    protected abstract int engineGetMacLength(  );
    protected abstract void engineInit(Key, AlgorithmParameterSpec);
    protected abstract void engineReset(  );
    protected abstract void engineUpdate(byte[], int, int);
    protected abstract void engineUpdate(byte);
}
```

Class javax.crypto.NullCipher

As its name implies, null cipher is a cipher that does nothing. You can use it to test cryptographic programs. Since a null cipher performs no transformations, its ciphertext will be exactly the same as its plaintext. Note that it is not an engine; you simply instantiate it directly.

Class Definition

```
public class javax.crypto.NullCipher
        extends javax.crypto.Cipher {

        // Constructors
        public NullCipher(  );
}
```

See also

`Cipher`

Class javax.crypto.SealedObject

A sealed object is a container for another object. The contained object is serialized and then encrypted using a cipher. You can construct a sealed object using any serializable object and a cipher that is initialized for encryption. To decrypt the contained object, call the `getObject( )` method with a cipher that is initialized for decryption.

Class Definition

```
public class javax.crypto.SealedObject
        extends java.lang.Object
        implements java.io.Serializable {

        // Constructors
        public SealedObject(Serializable, Cipher);

        // Instance Methods
        public final String getAlgorithm(  );
        public final Object getObject(Cipher);
        public final Object getObject(Key);
        public final Object getObject(Key, String);
}
```

See also

`PublicKey`, *`PrivateKey`*

Interface javax.crypto.SecretKey

A secret key represents a key that is used with a symmetric cipher. This interface is used strictly for type identification.

Interface Definition

```
public interface javax.crypto.SecretKey
        implements java.security.Key {
}
```

See also

Key

Class javax.crypto.SecretKeyFactory

A secret key factory is used to convert between secret key data formats; like a key factory, this is typically used to import a key based on its external format or to export a key to its encoded format or algorithm parameters. Instances of this class are obtained by calling the `getInstance( )` method. Keys may be exported by using the `translateKey( )` method; they are imported by using the `generate Secret( )` method.

Class Definition

```
public class javax.crypto.SecretKeyFactory
        extends java.lang.Object {

        // Constructors
        protected SecretKeyFactory(SecretKeyFactorySpi, Provider);

        // Class Methods
        public static final SecretKeyFactory getInstance(String);
        public static final SecretKeyFactory getInstance(String, String);

        // Instance Methods
        public final SecretKey generateSecret(KeySpec);
        public final KeySpec getKeySpec(SecretKey, Class);
        public final String getAlgorithm(  );
        public final Provider getProvider(  );
        public final SecretKey translateKey(SecretKey);
}
```

See also

KeySpec, *Provider*, *SecretKey*, *SecretKeyFactorySpi*

Class javax.crypto.SecretKeyFactorySpi

This class is the service provider interface for the `SecretKeyFactory` class. To create a secret key factory, make a subclass of this class and register your implementation with an appropriate provider. Because it is a JCE engine, it must be deployed in a specially signed jar file.

Class Definition

```
public abstract class javax.crypto.SecretKeyFactorySpi
        extends java.lang.Object {

        // Constructors
        public SecretKeyFactorySpi(  );

        // Protected Instance Methods
        protected abstract SecretKey engineGenerateSecret(KeySpec);
        protected abstract KeySpec engineGetKeySpec(SecretKey, Class);
        protected abstract SecretKey engineTranslateKey(SecretKey);
}
```

See also

`KeySpec`, *`Provider`*, *`SecretKey`*, *`SecretKeyFactory`*

F.6 Package javax.crypto.interfaces

Interface javax.crypto.interfaces.DHKey

This interface represents a public or private key used by the Diffie−Hellman key agreement implementation.

Interface Definition

```
public interface javax.crypto.interfaces.DHKey {

        // Instance Methods
        public abstract DHParameterSpec getParams(  );
}
```

See also

`DHPrivateKey`, *`DHPublicKey`*

Interface javax.crypto.interfaces.DHPrivateKey

This interface represents a private key in a Diffie−Hellman key agreement protocol.

Interface Definition

```
public interface javax.crypto.interfaces.DHPrivateKey
  implements javax.crypto.interfaces.DHKey, java.security.PrivateKey {

        // Instance Methods
        public abstract BigInteger getX(  );
}
```

See also

DHKey, *DHPublicKey*, *PrivateKey*

Interface javax.crypto.interfaces.DHPublicKey

This interface represents a public key in a Diffie−Hellman key agreement protocol.

Interface Definition

```
public interface javax.crypto.interfaces.DHPublicKey
  implements javax.crypto.interfaces.DHKey, java.security.PublicKey {

        // Instance Methods
        public abstract BigInteger getY(  );
}
```

See also

DHKey, *DHPrivateKey*, *PublicKey*

F.7 Package javax.crypto.spec

Class javax.crypto.spec.DESKeySpec

This class represents a key specification for DES keys; this specification may be used with a secret key factory to import and export DES keys.

Class Definition

```
public class javax.crypto.spec.DESKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        //Constants
        public static final int DES_KEY_LEN;

        // Constructors
        public DESKeySpec(byte[]);
        public DESKeySpec(byte[], int);

        // Class Methods
        public static boolean isParityAdjusted(byte[], int);
        public static boolean isWeak(byte[], int);

        // Instance Methods
        public byte[] getKey(  );
}
```

See also

SecretKeyFactory

Class javax.crypto.spec.DESedeKeySpec

This class represents a DESede key specification. It can be used with a secret key factory to import and export DESede keys.

Class Definition

```
public class javax.crypto.spec.DESedeKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        //Constants
        public static final int DES_EDE_KEY_LEN;

        // Constructors
        public DESedeKeySpec(byte[]);
        public DESedeKeySpec(byte[], int);

        // Class Methods
        public static boolean isParityAdjusted(byte[], int);

        // Instance Methods
        public byte[] getKey(  );
}
```

See also

SecretKeyFactory

Class javax.crypto.spec.DHGenParameterSpec

Instances of this class may be used to supply the algorithm−specific initialization method for generating Diffie−Hellman keys.

Class Definition

```
public class javax.crypto.spec.DHGenParameterSpec
        extends java.lang.Object
        implements java.security.spec.AlgorithmParameterSpec {

        // Constructors
        public DHGenParameterSpec(int, int);

        // Instance Methods
        public int getExponentSize(  );
        public int getPrimeSize(  );
}
```

See also

AlgorithmParameterGenerator, *AlgorithmParameterSpec*

Class javax.crypto.spec.DHParameterSpec

This class encapsulates the public parameters used in the Diffie−Hellman key agreement protocol. Instances of this class can be passed to the algorithm−specific initialization methods of a key pair generator.

Class Definition

```
public class javax.crypto.spec.DHParameterSpec
        extends java.lang.Object
        implements java.security.spec.AlgorithmParameterSpec {

        // Constructors
        public DHParameterSpec(BigInteger, BigInteger);
        public DHParameterSpec(BigInteger, BigInteger, int);

        // Instance Methods
        public BigInteger getG(  );
        public int getL(  );
        public BigInteger getP(  );
}
```

See also

AlgorithmParameterSpec, *KeyPairGenerator*

Class javax.crypto.spec.DHPrivateKeySpec

This class represents a key specification for Diffie−Hellman private keys. It can be used with a key factory to import and export Diffie−Hellman keys.

Class Definition

```
public class javax.crypto.spec.DHPrivateKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        // Constructors
        public DHPrivateKeySpec(BigInteger, BigInteger, BigInteger);
        public DHPrivateKeySpec(BigInteger, BigInteger, BigInteger, int);

        // Instance Methods
        public BigInteger getG(  );
        public int getL(  );
        public BigInteger getP(  );
        public BigInteger getX(  );
}
```

See also

DHParameterSpec, *DHPublicKeySpec*, *KeySpec*

Class javax.crypto.spec.DHPublicKeySpec

This class represents a key specification for Diffie−Hellman public keys. It can be used with a key factory to import and export Diffie−Hellman keys.

Class Definition

```
public class javax.crypto.spec.DHPublicKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        // Constructors
        public DHPublicKeySpec(BigInteger, BigInteger, BigInteger);
        public DHPublicKeySpec(BigInteger, BigInteger, BigInteger, int);

        // Instance Methods
        public BigInteger getG(  );
        public int getL(  );
        public BigInteger getP(  );
        public BigInteger getY(  );
}
```

See also

DHParameterSpec, *DHPrivateKeySpec*, *KeySpec*

Class javax.crypto.spec.IvParameterSpec

This class represents an IV (initialization vector) for a cipher using a feedback mode. Ciphers in CBC, PCBC, CFB, and OFB modes need to be initialized with an IV.

Class Definition

```
public class javax.crypto.spec.IvParameterSpec
        extends java.lang.Object
        implements java.security.spec.AlgorithmParameterSpec {

        // Constructors
        public IvParameterSpec(byte[]);
        public IvParameterSpec(byte[], int, int);

        // Instance Methods
        public byte[] getIV(  );
}
```

See also

AlgorithmParameterSpec, *Cipher*

Class javax.crypto.spec.PBEKeySpec

This class represents a key specification for a key that is used with passphrase encryption.

Class Definition

```
public class javax.crypto.spec.PBEKeySpec
        extends java.lang.Object
        implements java.security.spec.KeySpec {

        // Constructors
        public PBEKeySpec(String);

        // Instance Methods
        public final String getPassword(  );
}
```

See also

PBEParameterSpec, *SecretKey*, *SecretKeyFactory*

Class javax.crypto.spec.PBEParameterSpec

This class encapsulates the salt and iteration count that are used in passphrase−based encryption.

Class Definition

```
public class javax.crypto.spec.PBEParameterSpec
        extends java.lang.Object
        implements java.security.spec.AlgorithmParameterSpec {

        // Constructors
        public PBEParameterSpec(byte[], int);

        // Instance Methods
        public int getIterationCount(  );
        public byte[] getSalt(  );
}
```

See also

AlgorithmParameterSpec, *Cipher*, *PBEKeySpec*

Class javax.crypto.spec.RC2ParameterSpec

This class represents the parameters used to create RC2 keys.

Class Definition

```
public class javax.crypto.spec.RC2ParameterSpec
    extends java.lang.Object
    implements java.security.spec.AlgorithmParameterSpec {

    // Constructors
    public RC2ParameterSpec(int, byte[], int);
    public RC2ParameterSpec(int, byte[]);
    public RC2ParameterSpec(int);

    // Instance Methods
    public boolean equals(Object);
    public int getEffectiveKeyBits(  );
    public byte[] getIV(  );
    public int hashCode(  );
}
```

Class javax.crypto.spec.RC5ParameterSpec

This class represents the parameters used to create RC5 keys.

Class Definition

```
public class javax.crypto.spec.RC5ParameterSpec
    extends java.lang.Object
    implements java.security.spec.AlgorithmParameterSpec {

    // Constructors
    public RC5ParameterSpec(int, int, int, byte[], int);
    public RC5ParameterSpec(int, int, int);
    public RC5ParameterSpec(int, int, int, byte[]);

    // Instance Methods
    public boolean equals(Object);
    public byte[] getIV(  );
    public int getRounds(  );
    public int getVersion(  );
    public int getWordSize(  );
    public int hashCode(  );
}
```

Class javax.crypto.spec.SecretKeySpec

This class provides the specifications to key factories to translate secret keys.

Class Definition

```
public class javax.crypto.spec.SecretKeySpec
    extends java.lang.Object
    implements java.security.spec.KeySpec, javax.crypto.SecretKey {
```

```
    // Constructors
    public SecretKeySpec(byte[], String);
    public SecretKeySpec(byte[], int, int, String);

    // Instance Methods
    public boolean equals(Object);
    public String getAlgorithm(  );
    public byte[] getEncoded(  );
    public String getFormat(  );
    public int hashCode(  );
}
```

F.8 Package javax.net

This package contains various socket−related classes. Though it is used with SSL, the classes in this package are generic and can handle sockets of any protocol.

Class javax.net.ServerSocketFactory

This class is used to obtain server sockets. The default implementation of this method (returned from the `getDefault( )` method) will provide standard TCP server sockets; subclasses of this class can be used to obtain SSL server sockets.

Class Definition

```
public abstract class javax.net.ServerSocketFactory
    extends java.lang.Object {

    // Constructors
    protected ServerSocketFactory(  );

    // Class Methods
    public static ServerSocketFactory getDefault(  );

    // Instance Methods
    public abstract ServerSocket createServerSocket(int,
                                            int, InetAddress);
    public abstract ServerSocket createServerSocket(int, int);
    public abstract ServerSocket createServerSocket(int);
}
```

See also

`SSLServerSocketFactory`

Class javax.net.SocketFactory

Instances of this class can be used to create client sockets. The default factory (returned from the `getDefault( )` method) will create standard TCP sockets; subclasses of this class can be used to obtain SSL sockets.

Class Definition

```
public abstract class javax.net.SocketFactory
    extends java.lang.Object {

    // Constructors
    protected SocketFactory(  );

    // Class Methods
    public static SocketFactory getDefault(  );

    // Instance Methods
    public abstract Socket createSocket(InetAddress, int);
    public abstract Socket createSocket(InetAddress, int,
                                        InetAddress, int);
    public abstract Socket createSocket(String, int, InetAddress, int);
    public abstract Socket createSocket(String, int);
}
```

See also

SSLSocketFactory

F.9 Package javax.net.ssl

Classes in this package provide SSL and HTTPS implementations.

Class javax.net.ssl.HandshakeCompletedEvent

Instances of this class are sent to all registered listeners on an SSL socket when the protocol negotiation of the socket has concluded.

Class Definition

```
public class javax.net.ssl.HandshakeCompletedEvent
    extends java.util.EventObject {

    // Constructors
    public HandshakeCompletedEvent(SSLSocket, SSLSession);

    // Instance Methods
    public String getCipherSuite(  );
    public javax.security.cert.X509Certificate[]
                               getPeerCertificateChain(  );
    public SSLSession getSession(  );
    public SSLSocket getSocket(  );
}
```

See also

HandshakeCompletedListener, *SSLSession*

Interface javax.net.ssl.HandshakeCompleted−Listener

Classes that implement this interface can be registered with an SSL socket and will be notified when the socket has completed its protocol negotiation.

Class Definition

```
public interface javax.net.ssl.HandshakeCompletedListener
    implements java.util.EventListener {

    // Instance Methods
    public abstract void handshakeCompleted(HandshakeCompletedEvent);
}
```

See also

`HandshakeCompletedEvent`, *`SSLSession`*

Class javax.net.ssl.SSLServerSocket

Instances of this class are obtained from an `SSLServerSocketFactory` object and provide the server side of an SSL connection. The socket returned by the `accept( )` method of this class will be an SSL socket. Before calling the `accept( )` method, you can enable or disable the various cipher suites you want to support; the server and client sockets will negotiate those cipher suites transparently.

Class Definition

```
public abstract class javax.net.ssl.SSLServerSocket
    extends java.net.ServerSocket {

    // Constructors
    protected SSLServerSocket(int, int);
    protected SSLServerSocket(int, int, InetAddress);
    protected SSLServerSocket(int);

    // Instance Methods
    public abstract boolean getEnableSessionCreation(  );
    public abstract String[] getEnabledCipherSuites(  );
    public abstract boolean getNeedClientAuth(  );
    public abstract String[] getSupportedCipherSuites(  );
    public abstract boolean getUseClientMode(  );
    public abstract void setEnableSessionCreation(boolean);
    public abstract void setEnabledCipherSuites(String[]);
    public abstract void setNeedClientAuth(boolean);
    public abstract void setUseClientMode(boolean);
}
```

See also

`SSLSocket`, *`SSLServerSocketFactory`*

Class javax.net.ssl.SSLServerSocketFactory

Instances of this factory create server sockets that use SSL. The factories themselves are obtained via the `getDefault( )` method.

Class Definition

```
public abstract class javax.net.ssl.SSLServerSocketFactory
    extends javax.net.ServerSocketFactory {

    // Constructors
    protected SSLServerSocketFactory(  );

    // Class Methods
    public static synchronized ServerSocketFactory getDefault(  );

    // Instance Methods
    public abstract String[] getDefaultCipherSuites(  );
    public abstract String[] getSupportedCipherSuites(  );
}
```

See also

`SSLServerSocket, SSLSocketFactory`

Interface javax.net.ssl.SSLSession

SSL sessions are used to obtain information about a particular SSL client or server socket. In particular, the session object can be used to retrieve (via the `getPeerCertificateChain( )` method) the certificate of the connected peer; you should examine the name within this certificate to make sure that it matches the name of the server to which you want to connect. The session object is obtained from the SSL socket.

Class Definition

```
public interface javax.net.ssl.SSLSession {

    // Instance Methods
    public abstract String getCipherSuite(  );
    public abstract long getCreationTime(  );
    public abstract byte[] getId(  );
    public abstract long getLastAccessedTime(  );
    public abstract javax.security.cert.X509Certificate[]
                                    getPeerCertificateChain(  );
    public abstract String getPeerHost(  );
    public abstract SSLSessionContext getSessionContext(  );
    public abstract Object getValue(String);
    public abstract String[] getValueNames(  );
    public abstract void invalidate(  );
    public abstract void putValue(String, Object);
    public abstract void removeValue(String);
}
```

Class javax.net.ssl.SSLSessionBindingEvent

Events of this class are sent to all registered listeners on an SSL session when a new property is bound into that session.

Class Definition

```
public class javax.net.ssl.SSLSessionBindingEvent
    extends java.util.EventObject {

    // Constructors
    public SSLSessionBindingEvent(SSLSession, String);

    // Instance Methods
    public String getName(  );
    public SSLSession getSession(  );
}
```

See also

`SSLSession`, *`SSLSessionBindingListener`*

Interface javax.net.ssl.SSLSessionBindingListener

Classes that implement this interface can be registered with an SSL session and will be notified whenever a new property is bound into that session.

Class Definition

```
public interface javax.net.ssl.SSLSessionBindingListener
    implements java.util.EventListener {

    // Instance Methods
    public abstract void valueBound(SSLSessionBindingEvent);
    public abstract void valueUnbound(SSLSessionBindingEvent);
}
```

See also

`SSLSession`, *`SSLSessionBindingEvent`*

Interface javax.net.ssl.SSLSessionContext

Classes that implement this interface can be used to group together SSL sessions, which can be set or retrieved via their session ID.

Interface Definition

```
public interface javax.net.ssl.SSLSessionContext {

    // Instance Methods
    public abstract Enumeration getIds(  );
    public abstract SSLSession getSession(byte[]);
}
```

Class javax.net.ssl.SSLSocket

This class represents one peer in an SSL connection. Client side instances of this class are retrieved from an `SSLSocketFactory` instance; servers obtain instances of this class from the `accept( )` method of an SSL server socket.

Class Definition

```
public abstract class javax.net.ssl.SSLSocket
    extends java.net.Socket {

    // Constructors
    protected SSLSocket(String, int);
    protected SSLSocket(String, int, InetAddress, int);
    protected SSLSocket(InetAddress, int);
    protected SSLSocket(InetAddress, int, InetAddress, int);
    protected SSLSocket(  );

    // Instance Methods
    public abstract void addHandshakeCompletedListener(
                             HandshakeCompletedListener);
    public abstract boolean getEnableSessionCreation(  );
    public abstract String[] getEnabledCipherSuites(  );
    public abstract boolean getNeedClientAuth(  );
    public abstract SSLSession getSession(  );
    public abstract String[] getSupportedCipherSuites(  );
    public abstract boolean getUseClientMode(  );
    public abstract void removeHandshakeCompletedListener(
                            HandshakeCompletedListener);
    public abstract void setEnableSessionCreation(boolean);
    public abstract void setEnabledCipherSuites(String[]);
    public abstract void setNeedClientAuth(boolean);
    public abstract void setUseClientMode(boolean);
    public abstract void startHandshake(  );
}
```

See also

`SSLServerSocket`, *`SSLSocketFactory`*

Class javax.net.ssl.SSLSocketFactory

Instances of this class are used to obtain SSL sockets connected to particular servers and port numbers. The `getDefault( )` method is used to obtain the factory.

Class Definition

```
public abstract class javax.net.ssl.SSLSocketFactory
    extends javax.net.SocketFactory {

    // Constructors
    public SSLSocketFactory(  );

    // Class Methods
    public static synchronized SocketFactory getDefault(  );

    // Instance Methods
    public abstract Socket createSocket(Socket, String, int, boolean);
    public abstract String[] getDefaultCipherSuites(  );
    public abstract String[] getSupportedCipherSuites(  );
}
```

See also

`SSLSocket`, *`SSLServerSocketFactory`*

F.10 Package javax.security.auth

This package contains the classes that define JAAS.

Class javax.security.auth.AuthPermission

This class represents permission to interact with the `javax.security.auth.Policy`, `Subject`, `LoginContext`, and `Configuration` classes. It is a basic permission, so it accepts a name but no actions.

The valid names of this class are:

doAs
: Allow calling the `Subject.doAs( )` method.

doAsPrivileged
: Allow calling the `Subject.doAsPrivileged( )` method.

getSubject
: Allow calling the `Subject.getSubject( )` method.

getSubjectFromDomainCombiner
: Allow retrieving a subject that has a domain combiner.

setReadOnly
: Allow calling the `setReadOnly( )` method of the `Subject` class.

modifyPrincipals

Allow modifying the set of principals returned by the `getPrincipals( )` method of the `Subject` class.

modifyPublicCredentials
Allow modifying the public credentials of those principals.

getPolicy
Allow calling the `getPolicy( )` method of the `javax.security.auth.Policy` class.

setPolicy
Allow calling the `setPolicy( )` method of the `javax.security.auth.Policy` class.

refreshPolicy
Allow calling the `refresh( )` method of the `javax.security.auth.Policy` class.

refreshCredential
Allow calling the `refresh( )` method of a `Refreshable` object.

destroyCredential
Allow calling the `destroy( )` method of a `Destroyable` object.

createLoginContext
Allow code to instantiate a `LoginContext` object.

getLoginConfiguration
Allow calling the `getConfiguration( )` method of the `Configuration` class.

setLoginConfiguration
Allow calling the `setConfiguration( )` method of the `Configuration` class.

refreshLoginConfiguration
Allow calling the `refresh( )` method of the `Configuration` class.

Class Definition

```
public final class javax.security.auth.AuthPermission
    extends java.security.BasicPermission {

    // Constructors
    public AuthPermission(String, String);
    public AuthPermission(String);
}
```

Interface javax.security.auth.Destroyable

Classes (usually `Subjects`) that implement this interface allow their contents to be destroyed. This is useful if you want to reuse the object or don't want to hold sensitive information about the object in memory.

Interface Definition

```
public interface javax.security.auth.Destroyable {
    // Instance Methods
    public abstract void destroy(  );
    public abstract boolean isDestroyed(  );
}
```

See also

Subject

Class javax.security.auth.Policy

This class provides the interface to the JAAS policy. The default policy is implemented by the class specified as the `auth.policy.provider` property in the *java.security* file. You can write a new implementation of this class and install it by setting that property or by calling the `setPolicy( )` method of this class.

Class Definition

```
public abstract class javax.security.auth.Policy
    extends java.lang.Object {

    // Constructors
    protected Policy(  );

    // Class Methods
    public static Policy getPolicy(  );
    public static void setPolicy(Policy);

    // Instance Methods
    public abstract PermissionCollection getPermissions(
                                        Subject, CodeSource);
    public abstract void refresh(  );
}
```

Class javax.security.auth.PrivateCredential−Permission

This class represents permission to access the private credentials of a subject. The name of this permission is the name of the subject that you want to access (e.g., "sdo"); specify an asterisk to match all names. The only valid action for this permission is "read" (but you must specify the action in the policy file).

Class Definition

```
public final class javax.security.auth.PrivateCredentialPermission
    extends java.security.Permission {

    // Constructors
    public PrivateCredentialPermission(String, String);

    // Instance Methods
    public boolean equals(Object);
    public String getActions(  );
    public String getCredentialClass(  );
    public String[][] getPrincipals(  );
    public int hashCode(  );
    public boolean implies(Permission);
    public PermissionCollection newPermissionCollection(  );
}
```

Interface javax.security.auth.Refreshable

Classes (usually `Subjects`) that implement this interface allow themselves to be refreshed (e.g., by rereading their data from a persistent store).

Interface Definition

```
public interface javax.security.auth.Refreshable {

    // Instance Methods
    public abstract boolean isCurrent(  );
    public abstract void refresh(  );
}
```

Class javax.security.auth.Subject

Subjects represent the state of an authenticated user. The user may have a set of public and/or private credentials, as well as other principal information. They are generally returned from the `LoginContext` class.

The `doAs( )` methods of this class allow the target code to be run with permission checking to ensure the target subject has been granted the appropriate permissions.

Class Definition

```
public final class javax.security.auth.Subject
    extends java.lang.Object
    implements java.io.Serializable {

    // Constructors
    public Subject(  );
    public Subject(boolean, Set, Set, Set);

    // Class Methods
    public static Object doAs(Subject, PrivilegedExceptionAction);
    public static Object doAs(Subject, PrivilegedAction);
    public static Object doAsPrivileged(Subject,
                             PrivilegedAction, AccessControlContext);
    public static Object doAsPrivileged(Subject,
                     PrivilegedExceptionAction, AccessControlContext);
    public static Subject getSubject(AccessControlContext);

    // Instance Methods
    public boolean equals(Object);
    public Set getPrincipals(  );
    public Set getPrincipals(Class);
    public Set getPrivateCredentials(Class);
    public Set getPrivateCredentials(  );
    public Set getPublicCredentials(  );
    public Set getPublicCredentials(Class);
    public int hashCode(  );
    public boolean isReadOnly(  );
    public void setReadOnly(  );
    public String toString(  );
}
```

Class javax.security.auth.SubjectDomainCombiner

This class allows the domain checking of the `doAs( )` method to be optimized by providing a combined set of permissions and domains for the target subject. It is normally an opaque object to the developer.

Class Definition

```
public class javax.security.auth.SubjectDomainCombiner
    extends java.lang.Object
    implements java.security.DomainCombiner {

    // Constructors
    public SubjectDomainCombiner(Subject);

    // Instance Methods
    public ProtectionDomain[] combine(ProtectionDomain[],
                                      ProtectionDomain[]);
    public Subject getSubject(  );
}
```

F.11 Package javax.security.auth.callback

This package contains the classes and interfaces that allow a login module to interact with the user, requesting information such as their ID and password.

Interface javax.security.auth.callback.Callback

This is the interface all callback objects share, used only for type identification.

Interface Definition

```
public interface javax.security.auth.callback.Callback {
}
```

Interface javax.security.auth.callback.Callback−Handler

If you want to perform callbacks to obtain information from the user, you must create a class that implements this interface and provide that class to the `LoginContext` object. Certain login modules require that you implement certain callbacks; check the login module documentation for more details. In the `handle( )` method, you must loop through the array, find out the actual type of each object, and then handle the type specifically.

Interface Definition

```
public abstract javax.security.auth.callback.CallbackHandler {

    // Instance Methods
    public abstract void handle(Callback[]);
}
```

Class javax.security.auth.callback.ChoiceCallback

Login modules that want the user to make a particular choice will create an instance of this object and pass it to the callback handler. Your callback handler should call the `setSelectedIndex( )` method to indicate which choice(s) the user made.

Class Definition

```
public class javax.security.auth.callback.ChoiceCallback
    extends java.lang.Object
    implements javax.security.auth.callback.Callback {

    // Constructors
    public ChoiceCallback(String, String[], int, boolean);

    // Instance Methods
    public boolean allowMultipleSelections(  );
    public String[] getChoices(  );
    public int getDefaultChoice(  );
    public String getPrompt(  );
    public int[] getSelectedIndexes(  );
    public void setSelectedIndex(int);
    public void setSelectedIndexes(int[]);
}
```

Class javax.security.auth.callback.Confirmation−Callback

Login modules that want the user to acknowledge a particular situation will create an instance of this object and pass it to the callback handler. Your callback handler should call the `setSelectedIndex( )` method to indicate which confirmation the user selected (yes/no/ok, etc.).

Class Definition

```
public class javax.security.auth.callback.ConfirmationCallback
    extends java.lang.Object
    implements javax.security.auth.callback.Callback {

    // Constants
    public static final int CANCEL;
    public static final int ERROR;
    public static final int INFORMATION;
    public static final int NO;
    public static final int OK;
    public static final int OK_CANCEL_OPTION;
    public static final int UNSPECIFIED_OPTION;
    public static final int WARNING;
    public static final int YES;
    public static final int YES_NO_CANCEL_OPTION;
    public static final int YES_NO_OPTION;

    // Constructors
    public ConfirmationCallback(int, String[], int);
    public ConfirmationCallback(String, int, int, int);
    public ConfirmationCallback(String, int, String[], int);
    public ConfirmationCallback(int, int, int);

    // Instance Methods
    public int getDefaultOption(  );
    public int getMessageType(  );
    public int getOptionType(  );
    public String[] getOptions(  );
    public String getPrompt(  );
    public int getSelectedIndex(  );
    public void setSelectedIndex(int);
}
```

Class javax.security.auth.callback.LanguageCallback

Login modules use this callback to determine what locale they are running in. Your callback handler should fill in the appropriate locale.

Class Definition

```
public class javax.security.auth.callback.LanguageCallback
    extends java.lang.Object
    implements javax.security.auth.callback.Callback {

    // Constructors
    public LanguageCallback(  );

    // Instance Methods
    public Locale getLocale(  );
    public void setLocale(Locale);
}
```

Class javax.security.auth.callback.NameCallback

Login modules use this callback to determine the user ID. Your callback handler should return the appropriate user ID by calling the `setName( )` method.

Class Definition

```
public class javax.security.auth.callback.NameCallback
    extends java.lang.Object
    implements javax.security.auth.callback.Callback {

    // Constructors
    public NameCallback(String, String);
    public NameCallback(String);

    // Instance Methods
    public String getDefaultName(  );
    public String getName(  );
    public String getPrompt(  );
    public void setName(String);
}
```

Class javax.security.auth.callback.Password−Callback

Login modules use this callback to determine the user's password. Your callback handler should return the appropriate password by calling the `setPassword( )` method. If the `isEchoOn( )` method returns `false`, then the password should not be echoed as the user types it.

Class Definition

```
public class javax.security.auth.callback.PasswordCallback
    extends java.lang.Object
    implements javax.security.auth.callback.Callback {

    // Constructors
    public PasswordCallback(String, boolean);

    // Instance Methods
    public void clearPassword(  );
    public char[] getPassword(  );
    public String getPrompt(  );
    public boolean isEchoOn(  );
    public void setPassword(char[]);
}
```

Class javax.security.auth.callback.TextInputCallback

Login modules use this callback to retrieve an arbitrary text string from the user. Your callback handler set the appropriate text by calling the `setText( )` method.

Class Definition

```
public class javax.security.auth.callback.TextInputCallback
    extends java.lang.Object
    implements javax.security.auth.callback.Callback {

    // Constructors
    public TextInputCallback(String, String);
    public TextInputCallback(String);

    // Instance Methods
    public String getDefaultText(  );
    public String getPrompt(  );
    public String getText(  );
    public void setText(String);
}
```

Class javax.security.auth.callback.TextOutput−Callback

Login modules use this callback to display an arbitrary message to the user.

Class Definition

```
public class javax.security.auth.callback.TextOutputCallback
    extends java.lang.Object
    implements javax.security.auth.callback.Callback {

    // Constants
    public static final int ERROR;
    public static final int INFORMATION;
    public static final int WARNING;

    // Constructors
    public TextOutputCallback(int, String);

    // Instance Methods
    public String getMessage(  );
    public int getMessageType(  );
}
```

F.12 Package javax.security.auth.login

Class javax.security.auth.login.AppConfiguration−Entry

This class represents a single login module for a particular application. The control flags indicate whether the module is required, optional, etc. This class is only used by developers who write their own `Configuration` class.

Class Definition

```
public class javax.security.auth.login.AppConfigurationEntry
    extends java.lang.Object {

    public static class LoginModuleControlFlag
        extends java.lang.Object {

        // Constants
        public static final LoginModuleControlFlag OPTIONAL;
        public static final LoginModuleControlFlag REQUIRED;
        public static final LoginModuleControlFlag REQUISITE;
        public static final LoginModuleControlFlag SUFFICIENT;

        // Instance Methods
        public String toString(  );
    }

    // Constructors
    public AppConfigurationEntry(String, LoginModuleControlFlag, Map);

    // Instance Methods
    public LoginModuleControlFlag getControlFlag(  );
    public String getLoginModuleName(  );
    public Map getOptions(  );
}
```

See also

Configuration

Class javax.security.auth.login.Configuration

This class parses the login configuration files and produces a set of entries that represents the information in those files. If you want to handle configuration differently, you can write a subclass of this class and install it as the default configuration class by setting the `login.configuration.provider` property in the *java.security* file.

Class Definition

```
public abstract class javax.security.auth.login.Configuration
    extends java.lang.Object {

    // Constructors
    protected Configuration(  );

    // Class Methods
    public static Configuration getConfiguration(  );
    public static void setConfiguration(Configuration);

    // Instance Methods
    public abstract AppConfigurationEntry[]
                            getAppConfigurationEntry(String);
    public abstract void refresh(  );
}
```

Class javax.security.auth.login.LoginContext

This class is used to authenticate a user. You create an instance of this class (perhaps supplying an appropriate callback handler and/or subject) and then invoke the `login( )` method to authenticate the user. If that succeeds, the user may be retrieved with the `getSubject( )` method; that subject is then generally passed to the `Subject.doAs( )` method.

Class Definition

```
public class javax.security.auth.login.LoginContext
    extends java.lang.Object {

    // Constructors
    public LoginContext(String, Subject, CallbackHandler);
    public LoginContext(String);
    public LoginContext(String, Subject);
    public LoginContext(String, CallbackHandler);

    // Instance Methods
    public Subject getSubject(  );
    public void login(  );
    public void logout(  );
}
```

F.13 Package javax.security.auth.spi

Interface javax.security.auth.spi.LoginModule

If you want to write your own login module, you implement this interface and then list the appropriate class in the login configuration file. The `login( )` method should authenticate the user (perhaps requiring a callback). However, since other modules may be in use, the user is not fully authenticated until all modules pass. When that happens, the `commit( )` method is called, at which point the module should add the appropriate `Subject` object to the set of principals. If one or more modules failed, then the `abort( )` method is called instead, at which point the module should clean up its internal state.

Interface Definition

```
public abstract interface javax.security.auth.spi.LoginModule {

    // Instance Methods
    public abstract boolean abort(  );
    public abstract boolean commit(  );
    public abstract void initialize(Subject, CallbackHandler,
                                Map, Map);
    public abstract boolean login(  );
    public abstract boolean logout(  );
}
```

F.14 Package javax.security.cert

This package contains definitions of certificates that are used by some JSSE classes. These certificates are the same as those contained in the `java.security.cert` package, but JSSE cannot rely on that package to be present when it runs in certain versions of Java 2 Micro Edition.

Class javax.security.cert.Certificate

This class represents a generic certificate within JSSE. To convert between this class and the `java.security.cert.Certificate` class, obtain the encoded bytes of this certificate and give them to a `CertificateFactory` object.

Class Definition

```
public abstract class javax.security.cert.Certificate
    extends java.lang.Object {

    // Constructors
    public Certificate(  );

    // Instance Methods
    public boolean equals(Object);
    public abstract byte[] getEncoded(  );
    public abstract PublicKey getPublicKey(  );
    public int hashCode(  );
    public abstract String toString(  );
    public abstract void verify(PublicKey, String);
    public abstract void verify(PublicKey);
}
```

See also

`java.security.cert.Certificate`

Class javax.security.cert.X509Certificate

This class represents an X509 Certificate for JSSE. To convert between this class and the `java.security.cert.X509Certificate` class, obtain the encoded bytes of this class and give them to a `CertificateFactory` object.

Class Definition

```
public abstract class javax.security.cert.X509Certificate
    extends javax.security.cert.Certificate {

    // Constructors
    public X509Certificate(  );

    // Class Methods
    public static final X509Certificate getInstance(byte[]);
    public static final X509Certificate getInstance(InputStream);

    // Instance Methods
    public abstract void checkValidity(  );
    public abstract void checkValidity(Date);
    public abstract Principal getIssuerDN(  );
    public abstract Date getNotAfter(  );
    public abstract Date getNotBefore(  );
    public abstract BigInteger getSerialNumber(  );
    public abstract String getSigAlgName(  );
    public abstract String getSigAlgOID(  );
    public abstract byte[] getSigAlgParams(  );
    public abstract Principal getSubjectDN(  );
    public abstract int getVersion(  );
}
```

See also

`java.security.cert.X509Certificate`

F.15 Package com.sun.net.ssl

This package contains classes that are specific to Sun's implementation of JSSE. Even though they are not standard Java classes, they must be used if you want to perform key and certificate handling or HTTPS host verification.

Interface com.sun.net.ssl.HostnameVerifier

Classes that implement this interface can be registered with an HTTPS connection. If the underlying HTTPS socket detects a name mismatch between the server to which it connects and the certificate that the server presents, the `verify( )` method will be called to allow the user to accept the connection anyway.

Interface Definition

```
public interface com.sun.net.ssl.HostnameVerifier {

    // Instance Methods
    public abstract boolean verify(String, String);
}
```

See also

`HttpsURLConnection`

Class com.sun.net.ssl.HttpsURLConnection

This class implements an HTTPS connection to a particular server. You can use the static methods of this class to set global parameters (cipher suites, etc.) that apply to all subsequent HTTPS connections, and you can obtain information about the connection directly from the object. It is used most often to set the hostname verifier or the cipher suite for the connection.

Class Definition

```
public abstract class com.sun.net.ssl.HttpsURLConnection
    extends java.net.HttpURLConnection {

    // Variables
    protected HostnameVerifier hostnameVerifier;
    protected SSLSocketFactory sslSocketFactory;

    // Constructors
    public HttpsURLConnection(URL);

    // Class Methods
    public static HostnameVerifier getDefaultHostnameVerifier(  );
    public static SSLSocketFactory getDefaultSSLSocketFactory(  );
    public static void setDefaultHostnameVerifier(HostnameVerifier);
    public static void setDefaultSSLSocketFactory(SSLSocketFactory);

    // Instance Methods
    public abstract String getCipherSuite(  );
    public HostnameVerifier getHostnameVerifier(  );
    public SSLSocketFactory getSSLSocketFactory(  );
    public abstract javax.security.cert.X509Certificate[]
                                    getServerCertificateChain(  );
    public void setHostnameVerifier(HostnameVerifier);
    public void setSSLSocketFactory(SSLSocketFactory);
}
```

See also

HostnameVerifier

Interface com.sun.net.ssl.KeyManager

Classes that implement this interface are used to manage which keys are used by an SSL server socket when it accepts a connection. Key managers are registered by the `init( )` method of the `SSLContext` class; Sun's implementation of that class actually requires an `X509KeyManager` object.

Interface Definition

```
public abstract interface com.sun.net.ssl.KeyManager {
}
```

See also

X509KeyManager, SSLContext

Class com.sun.net.ssl.KeyManagerFactory

This class can be used to obtain a key manager, which can be registered with an SSL server socket to determine which keys that socket uses when it accepts a connection. Once you have a factory (from the `getInstance( )` method), you must initialize it with the keystore you want to use; the key managers returned from the `getKeyManagers( )` method can then be used to initialize an SSL context object.

Class Definition

```
public class com.sun.net.ssl.KeyManagerFactory
    extends java.lang.Object {

    // Constructors
    protected KeyManagerFactory(KeyManagerFactorySpi,
                                Provider, String);

    // Class Methods
    public static final String getDefaultAlgorithm(  );
    public static final KeyManagerFactory getInstance(String, String);
    public static final KeyManagerFactory getInstance(String);
    public static final KeyManagerFactory getInstance(String,
                                                      Provider);
    // Instance Methods
    public final String getAlgorithm(  );
    public KeyManager[] getKeyManagers(  );
    public final Provider getProvider(  );
    public void init(KeyStore, char[]);
}
```

See also

`SSLContext`

Class com.sun.net.ssl.KeyManagerFactorySpi

If you want to provide your own mechanism for SSL server sockets to handle keys, you must provide an implementation of this class and register that class with a security provider.

Class Definition

```
public abstract class com.sun.net.ssl.KeyManagerFactorySpi
    extends java.lang.Object {

    // Constructors
    public KeyManagerFactorySpi(  );

    // Protected Instance Methods
    protected abstract KeyManager[] engineGetKeyManagers(  );
    protected abstract void engineInit(KeyStore, char[]);
}
```

Class com.sun.net.ssl.SSLContext

This class is used to affect the context in which an SSL socket is created; in particular, the `init( )` method allows you to provide the key and trust managers that will be used to provide and verify the credentials used in the SSL protocol negotiation. The key and trust managers must actually be classes that implement the `X509KeyManager` and `X509TrustManager` interfaces (unless you've installed a different `SSLContext` implementation).

Class Definition

```
public class com.sun.net.ssl.SSLContext
    extends java.lang.Object {

    // Constructors
    protected SSLContext(SSLContextSpi, Provider, String);

    // Class Methods
    public static SSLContext getInstance(String, String);
    public static SSLContext getInstance(String, Provider);
    public static SSLContext getInstance(String);

    // Instance Methods
    public final String getProtocol(  );
    public final Provider getProvider(  );
    public final SSLServerSocketFactory getServerSocketFactory(  );
    public final SSLSocketFactory getSocketFactory(  );
    public final void init(KeyManager[], TrustManager[], SecureRandom);
}
```

See also

`X509KeyManager`, *`X509TrustManager`*

Class com.sun.net.ssl.SSLContextSpi

If you want to provide your own definition of the SSL context class, you must provide an implementation of this class and register it with a security provider. That would allow you to use key and trust managers that are not X509 specific.

Class Definition

```
public abstract class com.sun.net.ssl.SSLContextSpi
    extends java.lang.Object {

    // Constructors
    public SSLContextSpi(  );

    // Protected Instance Methods
    protected abstract SSLServerSocketFactory
                               engineGetServerSocketFactory(  );
    protected abstract SSLSocketFactory engineGetSocketFactory(  );
    protected abstract void engineInit(KeyManager[],
                                       TrustManager[], SecureRandom);
}
```

Class com.sun.net.ssl.SSLPermission

This class is used to test for permissions within JSSE. You do not use it in programs, though you must specify it in the appropriate *java.policy* file if you want code to be able to perform certain operations. This is a basic permission; it requires a name but no actions.

The two valid names for this permission are "setHostnameVerifier" (needed to call the `setHostnameVerifier( )` method of the `HttpsURLConnection` class) and "getSSLSessionContext" (needed to call the `getSessionContext( )` method of Sun's implementation of the `SSLSocket` class).

Class Definition

```
public final class com.sun.net.ssl.SSLPermission
    extends java.security.BasicPermission {

    // Constructors
    public SSLPermission(String, String);
    public SSLPermission(String);
}
```

Interface com.sun.net.ssl.TrustManager

Classes that implement this interface can be used to determine which certificates are used to verify the identity of an SSL peer. Trust managers are registered via the `init( )` method of the `SSLContext` class; Sun's implementation of that class requires an `X509TrustManager` object.

Interface Definition

```
public abstract interface com.sun.net.ssl.TrustManager {
}
```

See also

`X509TrustManager`, *`X509KeyManager`*, *`SSLContext`*

Class com.sun.net.ssl.TrustManagerFactory

Instances of this class are used to obtain trust managers, which provide the certificates used to verify the identity of an SSL peer. The trust manager must be initialized with an appropriate keystore.

Class Definition

```
public class com.sun.net.ssl.TrustManagerFactory
    extends java.lang.Object {

    // Constructors
    protected TrustManagerFactory(TrustManagerFactorySpi,
                                  Provider, String);

    // Class Methods
    public static final String getDefaultAlgorithm(  );
    public static final TrustManagerFactory getInstance(
                                               String, String);
    public static final TrustManagerFactory getInstance(String);
    public static final TrustManagerFactory getInstance(
                                               String, Provider);

    // Instance Methods
    public final String getAlgorithm(  );
    public final Provider getProvider(  );
    public TrustManager[] getTrustManagers(  );
    public void init(KeyStore);
}
```

Class com.sun.net.ssl.TrustManagerFactorySpi

If you want to implement your own trust manager, you register an instance of this class with an appropriate security provider.

Class Definition

```
public abstract class com.sun.net.ssl.TrustManagerFactorySpi
    extends java.lang.Object {

    // Constructors
    public TrustManagerFactorySpi(  );

    // Protected Instance Methods
    protected abstract TrustManager[] engineGetTrustManagers(  );
    protected abstract void engineInit(KeyStore);
}
```

Interface com.sun.net.ssl.X509KeyManager

Implementations of this interface are used to select the keys that an SSL socket presents during protocol negotiation. If you want to provide a custom key manager, create a class that implements this interface. Then create a `KeyManagerFactorySpi` class that returns instances of that class and register the factory with an appropriate security provider. Note that the certificates used with this class are `java.security.cert.X509Certificate` objects.

Interface Definition

```
public interface com.sun.net.ssl.X509KeyManager
    implements com.sun.net.ssl.KeyManager {

    // Instance Methods
    public abstract String chooseClientAlias(String, Principal[]);
    public abstract String chooseServerAlias(String, Principal[]);
    public abstract X509Certificate[] getCertificateChain(String);
    public abstract String[] getClientAliases(String, Principal[]);
    public abstract PrivateKey getPrivateKey(String);
    public abstract String[] getServerAliases(String, Principal[]);
}
```

Interface com.sun.net.ssl.X509TrustManager

Classes that implement this interface are used to verify the certificates presented by an SSL peer during protocol negotiation. If you want to provide a custom trust manager, create a class that implements this interface, create a `TrustManagerFactorySpi` class that returns that class, and register the factory with an appropriate security provider. Note that the certificates used with this class are `java.security.cert.X509Certificate` objects.

Interface Definition

```
public interface com.sun.net.ssl.X509TrustManager
    implements com.sun.net.ssl.TrustManager {

    // Instance Methods
    public abstract X509Certificate[] getAcceptedIssuers(  );
    public abstract boolean isClientTrusted(X509Certificate[]);
    public abstract boolean isServerTrusted(X509Certificate[]);
}
```

F.16 Package com.sun.security.auth

This package contains implementations of JAAS principals and policies. Although it is specific to Sun's implementation, you need at least the names of these classes so that you can configure them into JAAS login configuration files.

Class com.sun.security.auth.NTDomainPrincipal

This represents the Windows NT domain that a subject has logged into. In a login configuration file, specifying this class requires that the user be in the given domain.

Class Definition

```
public class com.sun.security.auth.NTDomainPrincipal
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public NTDomainPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public String toString(  );
}
```

Class com.sun.security.auth.NTNumericCredential

This represents the Windows NT numeric ID that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given ID.

Class Definition

```
public class com.sun.security.auth.NTNumericCredential
    extends java.lang.Object {

    // Constructors
    public NTNumericCredential(int);

    // Instance Methods
    public boolean equals(Object);
    public int getToken(  );
    public int hashCode(  );
    public String toString(  );
}
```

Class com.sun.security.auth.NTSid

This represents the Windows NT sid that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given ID.

Class Definition

```
public class com.sun.security.auth.NTSid
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public NTSid(String);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public String toString(  );
}
```

Class com.sun.security.auth.NTSid

This is the superclass for Windows NT principals based on the sid of the user.

Class Definition

```
public class com.sun.security.auth.NTSid
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public NTSid(String);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public String toString(  );
}
```

Class com.sun.security.auth.NTSidDomainPrincipal

This represents the Windows NT domain sid that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given ID.

Class Definition

```
public class com.sun.security.auth.NTSidDomainPrincipal
    extends com.sun.security.auth.NTSid {

    // Constructors
    public NTSidDomainPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String toString(  );
}
```

Class com.sun.security.auth.NTSidGroupPrincipal

This represents the Windows NT group sid that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given ID.

Class Definition

```
public class com.sun.security.auth.NTSidGroupPrincipal
    extends com.sun.security.auth.NTSid {

    // Constructors
    public NTSidGroupPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String toString(  );
}
```

Class com.sun.security.auth.NTSidPrimaryGroup−Principal

This represents the Windows NT primary group sid that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given primary ID.

Class Definition

```
public class com.sun.security.auth.NTSidPrimaryGroupPrincipal
    extends com.sun.security.auth.NTSid {

    // Constructors
    public NTSidPrimaryGroupPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String toString(  );
}
```

Class com.sun.security.auth.NTSidUserPrincipal

This represents the Windows NT user sid that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given user ID.

Class Definition

```
public class com.sun.security.auth.NTSidUserPrincipal
    extends com.sun.security.auth.NTSid {

    // Constructors
    public NTSidUserPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String toString(  );
}
```

Class com.sun.security.auth.NTUserPrincipal

This represents the Windows NT user ID that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given user ID.

Class Definition

```
public class com.sun.security.auth.NTUserPrincipal
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public NTUserPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public String toString(  );
}
```

Class com.sun.security.auth.PolicyFile

This class is Sun's default implementation of the policy checking performed by the `Subject.doAs( )` method. It supplies the permissions for a given subject executing code from the given codebase. You may specify a different policy object in the *java.security* file or via the `javax.security.auth.Policy` class.

Class Definition

```
public class com.sun.security.auth.PolicyFile
    extends javax.security.auth.Policy {

    // Constructors
    public PolicyFile(  );

    // Instance Methods
    public PermissionCollection getPermissions(Subject, CodeSource);
    public synchronized void refresh(  );
}
```

See also

`javax.security.auth.Policy`

Interface com.sun.security.auth.PrincipalComparator

Classes that implement the `Principal` interface for use with a login module may also implement this interface to support group or role operations. Sun's implementation of the `javax.security.auth.Policy` class tests to see if principals implement this interface; if they do, it checks to see if one particular principal implies another. A Solaris subject that contains a principal with a UID of 0, for example, implies a Solaris principal with any other UID.

Interface Definition

```
public interface com.sun.security.auth.PrincipalComparator {

    // Instance Methods
    public boolean implies(Subject);
}
```

See also

`javax.security.auth.Policy. javax.security.auth.Subject`

Class com.sun.security.auth.SolarisNumericGroup−Principal

This represents the Solaris group ID that belongs to a subject. In a login configuration file, specifying this class requires that the user belong to the given group.

Class Definition

```
public class com.sun.security.auth.SolarisNumericGroupPrincipal
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public SolarisNumericGroupPrincipal(String, boolean);
    public SolarisNumericGroupPrincipal(long, boolean);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public boolean isPrimaryGroup(  );
    public long longValue(  );
    public String toString(  );
}
```

Class com.sun.security.auth.SolarisNumericUser−Principal

This represents the Solaris user ID that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given user ID.

Class Definition

```
public class com.sun.security.auth.SolarisNumericUserPrincipal
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public SolarisNumericUserPrincipal(long);
    public SolarisNumericUserPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public long longValue(  );
    public String toString(  );
}
```

Class com.sun.security.auth.SolarisPrincipal

This represents the Solaris user ID that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given user ID. The ID in this class is specified as a string (e.g., "sdo").

Class Definition

```
public class com.sun.security.auth.SolarisPrincipal
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public SolarisPrincipal(String);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public String toString(  );
}
```

Class com.sun.security.auth.X500Principal

This represents the X500 name that belongs to a subject. In a login configuration file, specifying this class requires that the user have the given X500 name.

Class Definition

```
public class com.sun.security.auth.X500Principal
    extends java.lang.Object
    implements java.security.Principal, java.io.Serializable {

    // Constructors
    public X500Principal(String);

    // Instance Methods
    public boolean equals(Object);
    public String getName(  );
    public int hashCode(  );
    public String toString(  );
}
```

F.17 Package com.sun.security.auth.login

Class com.sun.security.auth.login.ConfigFile

This class is Sun's implementation of the configuration parser used by the `javax.security.auth.login.Configuration` class. You could subclass this class and install it via the `setConfiguration( )` method of that class.

Class Definition

```
public class com.sun.security.auth.login.ConfigFile
    extends javax.security.auth.login.Configuration {

    // Constructors
    public ConfigFile(  );

    // Instance Methods
    public AppConfigurationEntry[] getAppConfigurationEntry(String);
    public void refresh(  );
}
```

See also

`Configuration`

F.18 Package com.sun.security.auth.module

This package contains platform−specific login modules supplied by Sun. They are rarely used by developers, but you must specify their names and properties in the JAAS login configuration file.

Class com.sun.security.auth.module.JndiLogin−Module

This class allows you to authenticate users via a JNDI database. To configure this login module, you must define (in the login configuration file) two properties: `user.provider.url` and `group.provider.url`. For example, to contact an LDAP server, you would set the first property to *ldap://LDAPServerHostName/LDAPName* where `LDAPName` is the entry in the LDAP directory. The LDAP server must store information according to the format defined in RFC 2307. This module also works with NIS (using URLs of the form *nis://NISServerHostName/NISDomain/user* and *nis://NISServerHostName/NISDomain/system/group*).

Class Definition

```
public interface com.sun.security.auth.module.JndiLoginModule {

    // Constants
    public final String USER_PROVIDER;
    public final String GROUP_PROVIDER;

    // Constructors
    public JndiLoginModule(  )
    // Instance Methods
    public abstract boolean abort(  );
    public abstract boolean commit(  );
    public abstract void initialize(Subject, CallbackHandler,
                                    Map, Map);
    public abstract boolean login(  );
    public abstract boolean logout(  );
}
```

Class com.sun.security.auth.module.NTLoginModule

This class allows you to authenticate users via their Windows NT login. It takes all its information from the user's environment to produce a set of NT principals.

Class Definition

```
public interface com.sun.security.auth.module.NTLoginModule {

    // Constructors
    public NTLoginModule(  )

    // Instance Methods
    public abstract boolean abort(  );
    public abstract boolean commit(  );
    public abstract void initialize(Subject, CallbackHandler,
                                    Map, Map);
    public abstract boolean login(  );
    public abstract boolean logout(  );
}
```

Class com.sun.security.auth.module.SolarisLogin−Module

This class allows you to authenticate users via their Solaris login. It takes all its information from the user's environment to produce a set of Solaris principals.

Class Definition

```
public interface com.sun.security.auth.module.NTLoginModule {

    // Constructors
    public SolarisLoginModule(  )

    // Instance Methods
    public abstract boolean abort(  );
    public abstract boolean commit(  );
    public abstract void initialize(Subject, CallbackHandler,
                                    Map, Map);
    public abstract boolean login(  );
    public abstract boolean logout(  );
}
```

F.19 Miscellaneous Packages

This section lists security−related classes that appear in miscellaneous packages: permission classes, class loaders, and security managers.

Class java.awt.AWTPermission

This class represents permission to perform windowing operations, like opening a top−level window or examining the event queue. This is a basic permission, so it has no actions.

Class Definition

```
public final class java.awt.AWTPermission
        extends java.security.BasicPermission {

        // Constructors
        public AWTPermission(String);
        public AWTPermission(String, String);
}
```

See also

BasicPermission, *Permission*

Class java.io.FilePermission

This class represents permission to read, write, delete, or execute files. The name encapsulated in this permission is the name of the file; the string "<<ALL_FILES>>" represents all files while an asterisk represents all files in a directory and a hyphen represents all files that descend from a directory. The actions for this permission are read, write, execute, and delete.

Class Definition

```
public final class java.io.FilePermission
        extends java.security.Permission
        implements java.io.Serializable {

        // Constructors
        public FilePermission(String, String);

        // Instance Methods
        public boolean equals(Object);
        public String getActions(  );
        public int hashCode(  );
        public boolean implies(Permission);
        public PermissionCollection newPermissionCollection(  );
}
```

See also

Permission

Class java.io.SerializablePermission

This class represents permission to perform specific operations during object serialization −− specifically, whether or not object substitution may occur during serialization. As with all basic permissions, there are no actions associated with this class, which has one valid name: "enableSubstitution."

Class Definition

```
public final class java.io.SerializablePermission
        extends java.security.BasicPermission {

        // Constructors
        public SerializablePermission(String);
        public SerializablePermission(String, String);
}
```

See also

BasicPermission, *Permission*

Class java.lang.ClassLoader

This class is the basis for loading a class dynamically in Java. For historical reasons, it appears in this package, but it is recommended that all new class loaders subclass the `SecureClassLoader` class in the `java.security` package instead of using this class. Loading a class explicitly may be done with the `loadClass( )` method of this class (though classes are usually simply loaded as needed).

Class Definition

```
public abstract class java.lang.ClassLoader
        extends java.lang.Object {

        // Constructors
        protected ClassLoader(  );
        protected ClassLoader(ClassLoader);

        // Class Methods
        public static ClassLoader getSystemClassLoader(  );
        public static URL getSystemResource(String);
        public static InputStream getSystemResourceAsStream(String);
        public static Enumeration getSystemResources(String);

        // Instance Methods
        public ClassLoader getParent(  );
        public URL getResource(String);
        public InputStream getResourceAsStream(String);
        public final Enumeration getResources(String);
        public Class loadClass(String);

        // Protected Instance Methods
        protected final Class defineClass(String, byte[], int, int);
        protected final Class defineClass(byte[], int, int);
        protected final Class defineClass(String, byte[], int, int,
                                                ProtectionDomain);
        protected Package definePackage(String, String, String, String,
                                  String, String, String, URL);
        protected Class findClass(String);
        protected String findLibrary(String);
        protected final Class findLoadedClass(String);
        protected Class findLocalClass(String);
        protected final Class findSystemClass(String);
        protected Package getPackage(String);
        protected Package[] getPackages(  );
        protected synchronized Class loadClass(String, boolean);
        protected final void resolveClass(Class);
        protected final void setSigners(Class, Object[]);
}
```

See also

SecureClassLoader, *URLClassLoader*

Class java.lang.RuntimePermission

This class represents permission to perform certain runtime operations, such as executing other programs. Like all basic permissions, runtime permissions have no actions.

Class Definition

```
public final class java.lang.RuntimePermission
        extends java.security.BasicPermission {

        // Constructors
        public RuntimePermission(String);
        public RuntimePermission(String, String);
}
```

See also

BasicPermission, *Permission*

Class java.lang.SecurityManager

This class forms the primary interface to the security model of the virtual machine; it is recommended for backward compatibility that access to that model occur through this class rather than by calling the access controller directly. However, most of the methods of this class simply call the access controller.

Class Definition

```
public class java.lang.SecurityManager
        extends java.lang.Object {

        // Variables
        protected boolean inCheck;

        // Constructors
        public SecurityManager(  );

        // Instance Methods
        public void checkAccept(String, int);
        public void checkAccess(Thread);
        public void checkAccess(ThreadGroup);
        public void checkAwtEventQueueAccess(  );
        public void checkConnect(String, int);
        public void checkConnect(String, int, Object);
        public void checkCreateClassLoader(  );
        public void checkDelete(String);
        public void checkExec(String);
        public void checkExit(int);
        public void checkLink(String);
        public void checkListen(int);
        public void checkMemberAccess(Class, int);
        public void checkMulticast(InetAddress);
        public void checkMulticast(InetAddress, byte);
        public void checkPackageAccess(String);
        public void checkPackageDefinition(String);
        public void checkPermission(Permission);
        public void checkPermission(Permission, Object);
        public void checkPrintJobAccess(  );
        public void checkPropertiesAccess(  );
        public void checkPropertyAccess(String);
        public void checkRead(FileDescriptor);
        public void checkRead(String);
        public void checkRead(String, Object);
        public void checkSecurityAccess(String);
        public void checkSetFactory(  );
        public void checkSystemClipboardAccess(  );
        public boolean checkTopLevelWindow(Object);
        public void checkWrite(FileDescriptor);
        public void checkWrite(String);
        public boolean getInCheck(  );
        public Object getSecurityContext(  );
        public ThreadGroup getThreadGroup(  );

        // Protected Instance Methods
        protected native int classDepth(String);
        protected native int classLoaderDepth(  );
        protected native ClassLoader currentClassLoader(  );
        protected Class currentLoadedClass(  );
        protected native Class[] getClassContext(  );
        protected boolean inClass(String);
        protected boolean inClassLoader(  );
}
```

See also

`AccessController`

Class java.lang.reflect.ReflectPermission

This class represents the ability to obtain information via object reflections; specifically, whether private and protected variables and methods may be accessed through object reflection. As with all basic permissions, this permission carries no actions; it has a single name: access.

Class Definition

```
public final class java.lang.reflect.ReflectPermission
        extends java.security.BasicPermission {

        // Constructors
        public ReflectPermission(String);
        public ReflectPermission(String, String);
}
```

See also

`BasicPermission`, *`Permission`*

Class java.net.NetPermission

This class represents the ability to work with multicast sockets and the ability to use the authenticator classes. As with all basic permissions, this class carries no actions.

Class Definition

```
public final class java.net.NetPermission
        extends java.security.BasicPermission {

        // Constructors
        public NetPermission(String);
        public NetPermission(String, String);
}
```

See also

`BasicPermission`, *`Permission`*

Class java.net.SocketPermission

This class represents the ability to work with certain sockets. The name of this permission is constructed from the hostname or IP address of the machine on the other end of the socket and the port number; either portion of the name is subject to wildcard matching. Valid actions for this class include connect, resolve, accept, and listen.

Class Definition

```
public final class java.net.SocketPermission
        extends java.security.Permission
        implements java.io.Serializable {

        // Constructors
        public SocketPermission(String, String);

        // Instance Methods
        public boolean equals(Object);
        public String getActions(  );
        public int hashCode(  );
        public boolean implies(Permission);
        public PermissionCollection newPermissionCollection(  );
}
```

See also

Permission

Class java.net.URLClassLoader

This class provides a concrete class loader that may be used to load classes from one or more URLs (either HTTP−based or file−based URLs). Since it is a secure class loader, classes loaded from a URL class loader will be fully integrated into the access controller's security model.

Class Definition

```
public class java.net.URLClassLoader
        extends java.security.SecureClassLoader {

        // Constructors
        public URLClassLoader(URL[], ClassLoader);
        public URLClassLoader(URL[]);
        public URLCLassLoader(URL[], ClassLoader, URLStreamHandlerFactory);

        // Class Methods
        public static URLClassLoader newInstance(URL[]);
        public static URLClassLoader newInstance(URL[], ClassLoader);

        // Instance Methods
        public URL findResource(String);
        public Enumeration findResources(String);
        public URL[] getURLs(  );

        // Protected Instance Methods
        protected void addURL(URL);
        protected Package definePackage(String, Manifest, URL);
        protected Class findClass(String);
        protected PermissionCollection getPermissions(CodeSource);
}
```

See also

`ClassLoader`, *`SecureClassLoader`*

Class java.rmi.RMISecurityManager

The RMI security manager provides a security manager that is suitable for many RMI servers. It provides the ability for RMI applications to make socket−based connections to each other and otherwise follows the default security manager implementation.

Class Definition

```
public class java.rmi.RMISecurityManager
        extends java.lang.SecurityManager {

        // Constructors
        public RMISecurityManager(  );

        // Instance Methods
}
```

See also

`SecurityManager`

Class java.rmi.server.RMIClassLoader

While not a traditional class loader, this class allows classes to be loaded via the same mechanics as a class loader: the `loadClass( )` method may be called to load a class explicitly, and this class will also be used to load all subsequent classes required by the target class. This class loader will only load classes from the URL specified by the `java.rmi.server.codebase` property. The internal class loader used by this class is a secure class loader, so the security model of the access controller will be used by classes loaded in this manner.

Class Definition

```
public class java.rmi.server.RMIClassLoader
        extends java.lang.Object {

        // Class Methods
        public static Object getSecurityContext(ClassLoader);
        public static Class loadClass(String);
        public static Class loadClass(String, String);
        public static Class loadClass(URL, String);
}
```

See also

ClassLoader, *SecureClassLoader*

Class java.util.PropertyPermission

This class represents the ability to read or write properties. The name of a property permission is the name of the property itself; the action for a property permission is either set or get.

Class Definition

```
public final class java.util.PropertyPermission
        extends java.security.BasicPermission {

        // Constructors
        public PropertyPermission(String, String);

        // Instance Methods
        public boolean equals(Object);
        public String getActions(  );
        public int hashCode(  );
        public boolean implies(Permission);
        public PermissionCollection newPermissionCollection(  );
}
```

See also

Permission