Abstract Data Types in Java

McGRAW-HILL
JAVA MASTERS TITLES

Boone, Barry *JAVA Certification Exam Guide for Programmers and Developers*, 0-07-913657-5

Chung, David *Component Java*, 0-07-913690-7

Jenkins, Michael S. *Abstract Data Types in Java*, 0-07-913270-7

Ladd, Scott Robert *Java Algorithms*, 0-07-913696-6

Morgenthal, Jeffrey *Building Distributed JAVA Applications*, 0-07-913679-6

Reynolds, Mark C. *Object-Oriented Programming in JAVA*, 0-07-913250-2

Rice, Jeffrey and Salisbury, Irving *Advanced JAVA 1.1 Programming*, 0-07-913089-5

Savit, Jeffrey; Wilcox, Sean; Jayaraman, Bhuvana; *Enterprise JAVA: Where, How, When (and When Not) to Apply Java in Client/Server Business Environments;* 0-07-057991-1

Siple, Matthew *The Complete Guide to Java Database Programming*, 0-07-913286-3

Venners, Bill, *Inside the JAVA Virtual Machine*, 0-07-913248-0

Abstract Data Types in Java

Michael S. Jenkins

McGraw-Hill
New York • San Francisco • Washington, D.C. • Auckland
Bogotá • Caracas • Lisbon • London • Madrid • Mexico City
Milan • Montreal • New Delhi • San Juan • Singapore
Sydney • Tokyo • Toronto

Library of Congress Cataloging-in-Publication Data

Jenkins, Michael S.
Abstract data types in Java / Michael S. Jenkins.
p. cm.
Includes index.
ISBN 0-07-913270-7
1. Java (Computer program language) 2. Abstract data types
(Computer science) I. Title.
QA76.73.J38J45 1998 97-30666
005.7' 3-dc21 CIP

McGraw-Hill
A Division of The McGraw-Hill Companies

1 2 3 4 5 6 7 8 9 0 DOC/DOC 9 0 2 1 0 9 8 7

PN 032936-2
PART OF
ISBN 0-07-913270-7

The sponsoring editor for this book was Judy Brief and the production supervisor was Tina Cameron. It was set in Vendome ICG by Douglas & Gayle, Limited

Printed and bound by R.R. Donnelley & Sons Company.

Contents

Acknowledgments

I would like to take a moment to express my appreciation to all of the people who helped me through this project.

A big "Thank you!" goes to all the people at McGraw-Hill, especially Judy Brief for helping me to figure out which end is up.

I would like to thank Jeff Rice for all the advice and direction he offered while reviewing the manuscript.

I would also like to express my appreciation for all of the support I received from my fellow Java devotees in the Project A team at the Chicago Board of Trade throughout the course of working on this book.

And, of course, my eternal gratitude goes to my lovely wife, Juliana, and our two beautiful children, Dana and David, for all the sacrifices they made in order to allow me the time to complete this project.

Introduction

Abstract Data Types in Java examines the design and development of the data structures required for meaningful application development, specifically in the Java programming language. With its numerous examples and exercises, this book is intended as both a resource for the programmer and as a collegiate text. *Abstract Data Types in Java* provides extensive analysis, explanation, and code examples in the Java programming language for the data structures explored. An incremental learning approach is used to facilitate the comprehension and retention of the material. Simpler, basic abstract types evolve into the more complex structures chapter by chapter. Each chapter closes with a summary of the important topics discussed as well as exercises designed to illustrate the points covered and to solidify the reader's understanding.

This book is written for the intermediate-level programmer and the college-level computer sciences student who are studying advanced programming concepts. As a programmer, a solid understanding of abstract data types is integral to the software-development process. This understanding includes the design, use, and implementation of these data structures. Any large-scale software development project will use at least some of these abstract types in its implementation. This book addresses these needs. Since the examples and exercises in the book are implemented in the relatively new and very popular Java programming language, this

book should also appeal to programmers migrating from the more established industry languages such as C and C++.

To make good use of this book, you should have a reasonable familiarity with the Java programming language and its syntax. C and C++ programmers should have little problem following the examples supplied. Non-programmers and beginning programmers may want to have a Java language reference manual at hand. Appendix A supplies a brief overview of the Java language and syntax.

Chapter Overview

Chapter 1, Basic Concepts

This chapter answers the question, "What are abstract data types?" The idea of using well-designed *abstract data types* (ADTs) to simplify the development

life cycle and to create reusable code is well established. This chapter covers the basics of designing and implementing ADTs in an object-oriented programming language. As a foundation to exploring data abstraction, we will take a look inside Java and explore some of the internal workings of the Java runtime system. Java *reference objects* will be explained. The passing of reference and value types as arguments and how each type of argument passing is used in the Java programming language will be discussed. Near the end of this chapter, exercises are provided to stimulate understanding in the use of reference objects.

Chapter 2, Error Handling and Exceptions

This chapter explains the importance of critical and non-critical error handling. The use of return values is contrasted with the use of exception handling. The Java Exception superclass is explored in detail as an example. The syntax and mechanics of Throwing and Catching Exceptions are briefly covered. Examples of how to extend the Exception class are given and explained. Exercises demonstrate the use of standard exceptions as well as how the use of customized exceptions can facilitate smooth software development.

Chapter 3, Arrays, Vectors, and Sorting

This chapter takes a brief look at the basics of array handling and explains the vector as a generic extensible array type. The treatment of arrays as objects in Java is discussed and examples are provided for the declaration and initialization of Java arrays. The reasons for using vectors instead of arrays are outlined, and examples are given on how to extend vectors to provide functionality not available in a standard array. One of the examples in the chapter shows us how to use a container class to extend the functionality of the basic Java vector. The Quick Sort algorithm is explained and a simple implementation is presented. The exercises near the end of this chapter include the development of a Sortable interface to create the **`SortedVector`** data type, which brings together these concepts.

Chapter 4, Hash Tables

This chapter examines the hash table. The hash table is a container that allows for quick and easy storage and retrieval of data that has a unique

key associated with it. The concepts of hash codes and hash methods are discussed in detail. A simple hash table class is defined from scratch to demonstrate the concepts. How and when to use hash tables are explained, and examples are given that use the core Java class Hashtable and its subclass, the Properties class. The chapter concludes with exercises that include the use of the Properties object to parse command-line arguments.

Chapter 5, Linked Lists

In this chapter we examine the linked list. Linked lists are container types that store collections of data in a sequential order. The concept of the generic data *node* is introduced and explained in this chapter. The standard linked list operations are covered in detail, and examples are given for simple add, insert, and delete methods. Array-based and non-array linked list implementations are examined and contrasted. List traversal is explained and implemented using the java.util.Enumeration interface.

Chapter 6, Circular and Doubly-Linked Lists

In this chapter a few of the extensions to the linked list class will be covered. Better super classes will be defined, and the examples will help provide the explanation and implementation of doubly-linked and circular-linked lists. The impact of performance and flexibility are explored in these more complex implementations. Integration of the previously developed quick sort is among the exercises presented at the end of the chapter.

Chapter 7, Stacks

This chapter takes a look at the stack as a specialized linked list. The built-in Java Stack object is used as an example of a Vector based stack. An analysis of the internals for the stack is provided and a non-Vector implementation is developed as a contrast. Exercises present an opportunity to look at uses of the stack.

Chapter 8, Queues

This chapter explores another specialization of the linked list, the queue. Queues are used in systems requiring message handling, event processing,

and the sharing of resources such as printers. Throughout the chapter we will be walking through the concepts behind, and the implementation of, a standard first-in/first-out queue. We will compare the queue storage container to the stacks covered in the last chapter and their last-in/first-out schema. We will once again take a look at Vector and non-Vector implementations of the queue in the examples and exercises we cover in the chapter.

Chapter 9, Simple Trees

In this chapter we explain the structure and use of simple rooted trees. Rooted trees are specialized storage containers that possess a single entry point and arrange the elements contained in a hierarchical fashion. We will draw a comparison between the tree structure and

traditional linked lists such as those we have covered in previous chapters. We will take a look at the mechanism behind tree traversal and how it differs from that of the linked list. We will also briefly discuss the use of an Interface to provide generic search and compare functionality to the tree.

Chapter 10, Binary Trees

This chapter expands concepts provided in Chapter 9 and explains the Binary Tree. A Binary Tree is implemented with a balanced tree structure to improve performance. The search algorithm is explained, implemented, and contrasted to the sequential search available in non-sorted linked lists.

Chapter 11, Multi-Way Trees

This chapter explains the structure of more complex tree types. We will expand on the binary trees we've covered so far and take a close look at a specific multi-way tree, the 2-3-4 tree. We will draw comparisons between the newly introduced multi-way trees and the binary tree structures we've looked at previously. Examples are provided to illustrate how a multi-way tree can be rendered as a binary implementation. Implementations of the tree types are walked through in the examples. Exercises encourage the development of other variations of the multi-way trees.

Chapter 12, B-Trees

In this chapter we will take a detailed look at the B-Tree data structure as an extension of the red/black and 2-3-4 trees. B-Trees are typically used to index large data sets and external data stores such as database files. We will take a look at a simple B-Tree implementation to help walk through the concepts presented in the chapter. Exercises include the development of a simple indexed data file.

About the Examples

All of the source code in this book was written using Version 1.1.3 of the Java Development Kit (JDK). All examples were tested and compiled using the JDK from Sun Microsystems on the Solaris™ and Windows 95™ platforms. Source code appears in a monospaced font (`Courier`).

All of the examples should be source compatible with any Version l.l.x of the JDK and should compile using any development environment that conforms to the Version 1.1 specification. With the exception of the inner classes used in the later sections of the book, all of the code should compile and run without modification using any previous versions of the JDK as well.

All of the source examples in this book are written using the same basic style. The following coding style guidelines are used to enhance source readability throughout the book:

- Class names always begin with a capital letter.
- Variable names begin with a lowercase letter with each subsequent word in the variable name beginning with a capital letter.

- Constants (**`public final static`**) are in all uppercase.
- In cases where two or more classes are defined in the same source file, the **`public`** class is defined first followed by classes of **`default`** or **`protected`** scope.
- "**`extends`**" and "**`implements`**" clauses are indented on lines subsequent to the class name definition.

- The methods of a class are defined before **`static(class)`** variables and instance variables.
- All curly braces ("{ " and "}") are vertically aligned.

```
public class Foo
    extends Bar
{
     public Foo( int argumentOne )
     {
         System.out.println( "Hello World" );
     }

     public final static int LEFT = 1;
     String myString;
}
```

Contacting the Author

Michael S. Jenkins is an independent software development consultant. For the past nine years he has been assisting his clients in successfully developing their business applications and enterprise systems. He has worked with companies such as the Chicago Board of Trade, the Chicago Stock Exchange, Baxter Healthcare, A.C. Nielsen/Dun & Bradstreet, and other major corporations.

If you have any questions or comments about anything in this book you can contact the author via email at:

`java@jcs-inc com`

or on the world wide web at the following URL:

`http://www.wwa.com/-mjenkins`

Chapter 1
Basic Concepts

This chapter answers the question, "What are abstract data types?" The idea of using

well-designed *abstract data types* (ADTs) to simplify the development life cycle and to create reusable code is well established. This chapter covers the basics of designing and implementing ADTs in an object-oriented programming language. As a foundation to exploring data abstraction, we will take a look inside Java and explore some of the internal workings of the Java runtime system. Java reference objects will be explained. The passing of reference and value types as arguments and how each type of argument passing is used in the Java programming language will be discussed. Near the end of this chapter, exercises are provided to stimulate understanding in the use of reference objects.

Abstract Data Types

This book is an introduction to abstract data types. So what are ADTs? To answer this question, we'll take a look at something we already know about: an integer data type. Virtually every modern programming language has some representation for an integer type.

In Java, we'll look at the primitive type *int*. An initialized Java int variable holds a 32-bit signed integer value between -2^{32} and 2^{32} -1. So, we've established that an int holds data.

Operations can be performed on an int. We can assign a value to an int. We can apply the Java unary prefix and postfix increment and decrement operators to an int. We can use an int in binary operation expressions, such as addition, subtraction, multiplication, and division. We can test the value of an int, and we can use an int in an equality expression.

In performing these operations on an int variable, the user does not need to be concerned with the implementation of the operation. The internal mechanism by which these operations work is irrelevant. Examine the following simple code fragment, for example:

```
int i = 0;
i++;
```

The user knows that after the second statement is executed, the value of the **`i`** variable is 1. It isn't important to know how the value became 1—just that in performing the increment in this example, **`i`** always will equal 1.

The user also does not need to know how the value is represented and stored internally Things such as byte order again are irrelevant to the user in the preceding code example.

To summarize the built-in int data type, an int does the following:

- An int holds an item of data.
- Predefined operations can be performed on an int.
- An int has an internal representation that is hidden from the user in performing these operations.

If we consider the primitive data types in this light, it is easy to understand the definition we will give to ADTs. An ADT is defined as the following:

- An ADT is an externally defined data type that holds some kind of data.[1]
- An ADT has built-in operations that can be performed on it or by it.
- Users of an ADT do not need to have any detailed information about the internal representation of the data storage or implementation of the operations.

So, in effect, an ADT is a data construct that encapsulates the same kinds of functionality that a built-in type would encapsulate. This does not necessarily imply that ADTs need to have addition or increment operations in order to be valid or useful, and it does not mean that any of the built-in operators will work with an ADT. It only means that the appropriate operations for the type created will be transparently available and that the user does not need to be concerned with the implementation details.

Classes and Abstract Data Types

In the Java language, all user-defined data types are classes. A *class* is a notation used by an object-oriented programming language to describe the layout and functionality of the objects that a program manipulates. All Java ADTs therefore are described by one or more classes. Not all classes are ADTs, but certainly all ADTs are implemented as classes. The built-in types in Java are not classes. This section takes a look at the differences among the various Java types.

Reference Objects and Value Types

In Java, two basic types of variables exist: primitive types and reference types. *Primitive types* are the standard built-in types we would expect to find in any modern programming language: int, long, short, byte, char,

---

[1]We will use the generic term *data* to refer to what, in most cases, will be a Java object.

boolean, void,[2] float, and double. *Reference types* are any variables that refer to an object. This is an important distinction, because the two variable types are treated differently in various situations. All reference type objects are of a specific class, for example.

All classes in Java are derived from the root class **`Object`**. So, given the rules of inheritance and polymorphism, an **`Object`** class variable can refer to any reference object of any class. In other words, a widening conversion can take place from any class to **`Object`**. Take a look at the following code, for example:

```
String s = new String("Hello World");
Object o = s;

MyClass m = new MyClass();
Object o = m;
```

One of Java's strengths is the fact that it uses a polymorphic model[3] wherein all classes are derived from a common root. All objects share a common base *Application Programming Interface* (API). This does not apply to the primitive types, however. The following code does

not work, for example:

```
double d = 3.8;
Object o = d;
```

The *Java Development Kit* (JDK) from Sun Microsystems comes complete with a set of classes defined by Sun as the *core* API. All these classes are the Java equivalent of a standard library. The Java core classes include *wrapper classes* for all the primitive types: **`Integer`** for int, **`Double`** for double, **`Float`** for float, and so on. Of course, all these wrapper classes are derived from the root class **`Object`** as well.

One unique case is that of the array. An *array* can be an array of primitives or an array of reference types. For the most part, the array itself is treated as a reference type of the **`Object`** class. The individual members, of course, are not treated in any special way. The rule of thumb is that anything created by the operator **`new`** is assignable to **`Object`.** So, again, the following code would work:

---

[2]Void is a valid return type, even though it technically is not a data type. Variables cannot be declared as type void. Void is used to denote that a method returns nothing.

[3]Polymorphism is an Object Oriented Programming term used to describe the capability of an object of one class to be treated as an object of another class due to the fact that the two classes maintain a hierarchical relationship.


```
long 1[10] = new long[10];
Object o = 1;

Object a[] = new Object[10];
Object o = a;
```

What does all this mean in terms of ADTs? Well, if we create a construct that works with objects of class **`Object`,** we can use the construct with any reference type in place of **`Object`**.

Passing Reference and Value Types

When calling a class member function, the developer will pass any required parameters to the method as arguments to the method call. Suppose that class `foo` has a member method declared as the following:

```
public int bar( String s, int i )
```

The caller of the method must supply a **`String`** (or an equivalent object that is automatically convertible to **`String`**) and an **`int`** to the call, or the compiler will generate an error. The questions here are, "What are we passing?" and "What are the consequences of passing any given parameter type?"

In very oversimplified terms, when a method is called, the system takes the arguments passed to the method from the calling routine and pushes them on the program stack. The execution point in the program then is *jumped* to the beginning of the method's code. The system then pops the arguments off the stack and uses them as variables of the types declared in the method's

parameter list. This type of mechanism enables methods to be passed arguments that normally may be outside the method's scope of visibility. When it is time for the method to return to the calling routine, it pushes the return value onto the stack. The program then jumps back to the calling routine and pops the return value back off the stack. For the purposes of this discussion, it is not important that we know the details of how a stack works. It is enough to know that a stack is a construct used to store data (see Figure 1-1). For more information on stacks, see Chapter 7, "Stacks."

Java uses a mechanism called *pass by value* to handle argument passing in method calls. This means that the system makes a copy of the value of the argument and pushes that onto the stack for the called method to access. In the following example, the value 4 is passed to the method **`foo()`**:

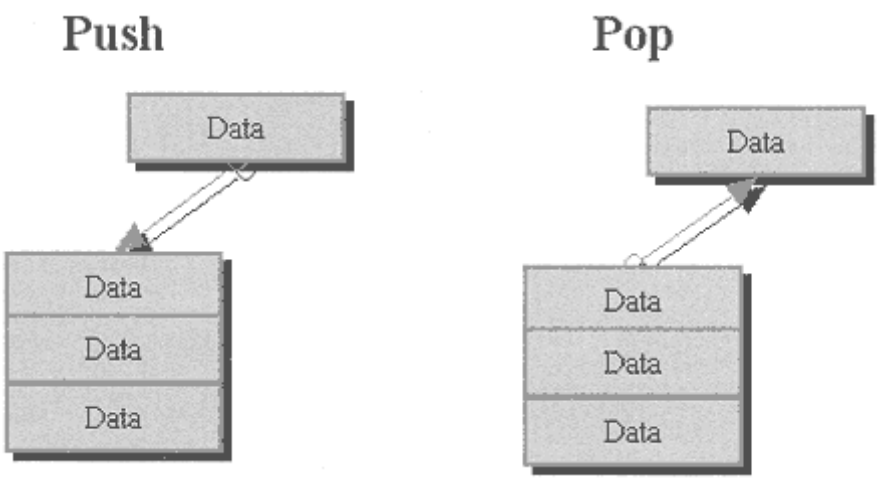


Figure 1-1
A typical data stack where one data item is "pushed" onto and then "popped" off of the stack.

```
int i = 4;
foo(i);
```

The method itself has no knowledge of the variable **`i`**. Changes made by **`foo()`** to the value passed will have no effect on **`i`** from the caller. If 4 is incremented to 5, for example, the value of **`i`** remains 4.

This pass by value approach is relatively straightforward for primitive types. But what about reference types? Aren't they references to objects? Isn't passing a reference equivalent to passing the original object itself? To answer these questions, take a closer look at the relationship between Java objects and the variables that are declared to hold them. Think about what really is happening in this statement:

```
String s = new String("Hello World");
```

Here, **`s`** is a variable of class **`String`**. The operator **`new`** allocates enough memory for a **`String`** object and calls the constructor for **`string`** with the argument **`"Hello World"`**. The return value for the operator **`new`** is a handle to the newly created **`String`** object. A *handle* to an object is basically an indicator to a location in memory. You might be familiar with *pointers* from the C and C++ programming languages. The handle is similar to a pointer; it

does "point" to an object. Unlike the more traditional pointers, though, a handle to a Java object cannot be modified except in the case of assignment to variables. A Java reference variable can be reassigned to a different object.

The implications of the differences between handles and pointers are subtle but important. When a reference type is passed as an argument to a method, the *handle* to the object is copied and passed—not the object itself So, in this code segment, the output would be **"Hello"**:

```
String s = new String( "Hello" );

change( s );
System.out.println( s );

. . .

public void change( String t )
{
     t = new String( "World" );
}
```

The handle to the object containing **"Hello"** is passed to **change()** as **String t. t** is reassigned to the new object containing **"World"**, but **s** remains unchanged. So, on the return of the function, **"World"** is left unreferenced, and the memory it occupies eventually is reclaimed by the garbage collector

So, any handle that we want to be reassigned during a method call must be the return value for the method, or the handle must be a member of an enclosing or wrapper class.

In the following example, a new string containing **"Hello"** is created:

```
String s = new String("Hello");
s = s.concat(" World");
```

When the **concat()** method then is used, a new string is created in the **concat()** method containing **"Hello World"** and is returned to the calling routine. This new string is completely unrelated to the original string **"Hello".** The **concat()** method is defined to return a **String** object.

In the next example, **StringWrapper** contains as a member field a **String** object:

```
class StringWrapper
{
     public String s;
}

. . .
```



```
changeString( StringWrapper t )
{
        t.s += " World";
```

```
}

StringWrapper s = new StringWrapper();
s.s = "Hello";
changeString(s);
```

Here, the **`StringWrapper`** object is passed as an argument to **`changeString()`**, and **`StringWrapper.s`** is reassigned to the new string "**`Hello World`**". After returning from the call to **`changeString()`**, the calling routine has access to the new "**`Hello World`**" string. A core class called **`StringBuffer`** provides a mutable **`String`** class. This class is much more complete than this simple example here.

Why Use Abstract Data Types?

Now that we have some idea of what ADTs are, this section takes a look at why we use them. The **`String`** class has been mentioned several times in this chapter. The **`String`** class provides a mechanism by which string literals may be stored, accessed, and manipulated. It provides methods with which we can compare, concatenate, copy, and convert strings. If a **`String`** class did not exist, string operations would have to be implemented from scratch each time they were needed.

A robust and reasonably generic **`String`** class gives us the capability to use these string operations at any time without having to "reinvent the wheel" each time. So ADTs provide us with code reusability. After we encapsulate the operations required to make a useful **`String`** class, we can reuse those facilities at any time in the future, with little or no additional development effort.

This also is the case with other ADTs, such as the ones we'll develop and examine in the following chapters. By designing our ADTs to be as generic as possible, we can reuse them in various situations and over several projects. Any time we develop an object or a group of related objects that can be reused, we reduce the overall development time of a project.

There are certain guidelines that need to be followed to make ADT's reusable. In this book, we are primarily concerned with *container* ADTs. A container object's primary purpose is to hold other objects. The contain-

ers we will design and implement in the following chapters will hold various types of data.

To make oour containers reusable, we need to make them generic. *Generic*, in this sense, means that the containers need to follow three rules:

1. Containers need to be able to store data of any kind.

2 Containers should provide a public interface that encompasses only behaviors that would be useful in a general sense.

3. Containers should be kept insulated from application-specific considerations.

To satisfy rule 1, we can select the **`Object`** class as our data type. This means that we will define our API for each of the ADTs to store and retrieve data of class **`Object`.** As discussed

earlier, the **`Object`** class is the root for all classes in Java. Therefore, any class defined in Java is assignable directly to a variable of class **`Object`.** If we were to specify a data class called **`MyDataClass`,** for example, we could pass a handle to that class to any method defined to take a parameter of the **`Object`** class. This is a standard, automatic widening conversion and requires no typecasting. The reverse narrowing conversion always requires casting. So extracting our data back out of the constructs requires a cast to the appropriate type. If a **`getData()`** method is defined to return a type **`Object`,** for example, we need to cast the returned handle back to **`MyDataClass`** explicitly.

As a brief aside, take a quick look at typecasting. Typecasting enables the programmer to temporarily change the type of an object or primitive. Two types of typecasting exist: automatic (or implicit) casting and manual (or explicit) casting. *Implicit casting* can be used when the compiler is able to determine that the type change is safe. *Explicit casting* is required if the safety of the typecast cannot be determined until runtime. To understand when each of the two types is appropriate, it is necessary to comprehend a little about the relationship of classes.

In the Java language, all classes are related in a hierarchical manner. The base of the hierarchy is the **`Object`** class. As discussed earlier, all classes in Java are derived from the root class **`Object`.** Each class that is subclassed directly from **`Object`** creates a new branch in the hierarchy. When these subclasses are subclassed, they extend and split the branch farther and farther. Figure 1-2 shows a sample hierarchy.

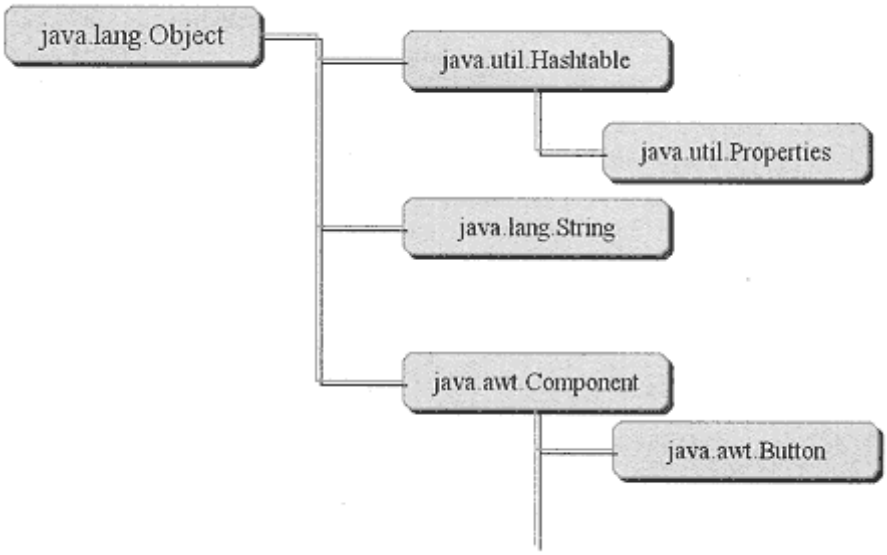


Figure 1-2
A partial inheritance diagram for the Java core classes.

As we can see, if we start tracing the hierarchy tree backward from any class on the tree, it eventually leads us back to the **`Object`** class. Each class along the route from the starting class is a superclass of the starting class as well as a superclass of its immediate child classes. Because a subclass extends its superclass, any subclassed object can be treated as a member of the superclass. This process is known as a *widening conversion.* An object's type is being widened toward the more general. The compiler can assume that any widening conversion is

safe; therefore, the compiler can automatically supply the conversion.

This process works fine as long as the conversion is in the direction of the superclass or more general case. Because Java is polymorphic, it is possible to determine at runtime the real type of an object. This runtime type information is necessary to determine whether a narrowing conversion is safe. Because a subclass is actually an extension of its parent class, there will naturally be a possibility that there is some field information in the subclass that isn't in the superclass.

Because this determination is done at runtime, this type of invalid cast cannot be detected at compile time. If it is detected at runtime, the system throws a runtime exception, the **`ClassCastException`**, to indicate the error. If the system throws this exception, the offending thread is halted. For a more detailed discussion of exceptions, see Chapter 2, "Error Handling and Exceptions."

We can create an object of type **`MyClass`**, as in the following example, and pass it to a method that takes an **`Object`** as an argument. The widening conversion is checked at compile time and therefore is implicitly done. If we then return the same object as a type **`Object`**, the narrowing conversion is checked at runtime and therefore needs to have an explicit cast.

```
MyClass m = new Myclass();

m = (MyClass)processData( m );

. . .

public Object processData(Object o)
{
     . . .
     return o;
}
```

When designing around rules 2 and 3, we need to keep in mind which behaviors are specific to the application we will be implementing and which are a function of the ADT itself Developers have a tendency to over-design container and utility classes to include every conceivable functionality into the class itself Generally, this is a mistake and is probably one of the biggest causes of code non-reusability. Keep in mind that we can subclass the ADT class and add case-specific code to the subclass. This leaves the base ADT class uncluttered and much more likely to be suitable for reuse.

Suppose that we design a base class called **`Polygon`**. We would restrict the fields and methods in the base class to those dealing with any generic polygon. We could have a **`numberOfSides`** field and accessor methods, but probably not an **`area()`** method, because the area calculations would be dependent on the specific polygon we instantiate. Then we could subclass **`Polygon`** into **`Rectangle`**, **`Quadrilatera`**l, **`Octagon`**, and so on. We also could subclass **`Rectangle`** into **`Square`** as a specific case of **`Rectangle`**. A sample hierarchy is shown in Figure 1-3.

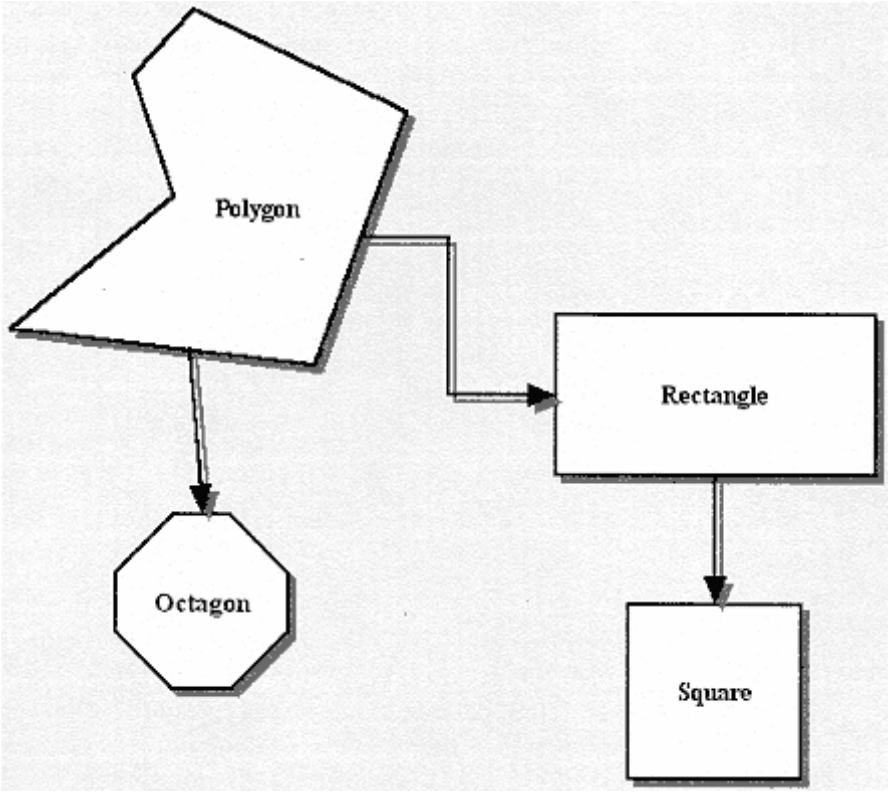


Figure 1-3
An example of inheritance providing specialization in a group of objects.

Exercises

1. Write a small Java application that uses the **`String`** class to input one word at a time and outputs a complete sentence when a terminating punctuation mark is entered.

2. Write an application similar to the one in exercise 1, but output the words in the sentence in reverse order.

3. Write the same application using a single **`StringBuffer`** object passed to the input method to collect the words in the sentence.

4. Write a class **`MutableInteger`** similar to the Java core class **`Integer`** but with the capability to change the value of the integer. (The **`Integer`** class, like the **`String`** class, is immutable after it is initialized.)

Summary

In this chapter, we learned the following:

- Abstract data types are similar to built-in types in that they have the same functionality.

- All ADTs in Java are implemented as classes.
- Primitive types and reference types have very different properties.
- All arguments to methods are passed by value in Java. Primitive types pass the value of the variable; reference types pass the value of the handle.
- Widening conversions of reference types passed as arguments are automatic while narrowing conversions require a type cast.
- Following a few simple design rules can promote code reusability, especially in the design of ADTs.

Chapter 2
Error Handling and Exceptions

This chapter explains the importance of critical and non-critical error handling. The use of return values is contrasted with the use of exception handling. The Java **`Exception`** superclass is explored in detail as an example. We'll take a brief look at the syntax and mechanics of throwing and catching exceptions. This chapter also provides examples of extending the **`Exception`** class. Near the end of this chapter, exercises demonstrate using standard and customized exceptions to facilitate smooth software development.

What Are Exceptions?

The proper handling of exception conditions is integral to sound software development. The first question that may come to mind is, "What is an exception?" An *exception*, in terms of software development, is an anomalous situation in which the state of the program is in jeopardy of becoming or has become unstable or corrupt.

One example of this condition is when a program is trying to call a non-static method for which the instance has not been defined or initialized. In Java, this state would generate a **`NullPointerException`**. In this case, the exception condition must be handled immediately to prevent the program from coming to an unexpected halt. We could use this code, for example:

```
public class ExceptionTest
{
     public static void main( String argv[] )
     {
          Vector v = null;
          try
          {
              v.elementAt(0);
```

```
            }
            catch( NullPointerException e )
            {
                System.out.println("Exception Handled");
            }
        }
    }
```

This code attempts to call the **elementAt()** method of **Vector** for instance v without first creating the object to which **v** refers. When an instance method is called from within a program, the Java runtime environment automatically passes the method the handle **this**. **this** is a reference to the calling instance object. If the **this** argument refers to **null**, a **NullPointerException** is thrown. After it is thrown, if the exception is not caught, the program is halted by the Java runtime environment. (We will cover the mechanics of throwing and catching exceptions shortly.) The point here is that once program execution reaches the **elementAt()** call, there is no way to continue processing. If the call were allowed to continue, the method would be accessing memory in an undefined location. This would not only be a security breach, but it also could cause problems with the runtime or the system itself

Return Values Versus Exceptions

Some programming languages do not support exception handling. As a matter of fact, exception handling such as that supported by Java is relatively new to mainstream software development. In the past, it was common to use the return value of a method or function to determine whether the call was successful, as shown in this example:

```
public boolean toUpper( String s )
{
    if( s == null )    // Error condition!
        return false;

    . . .     // process the String

    return true;
}
```

In this case, the method would return **true** or **false** to indicate whether the method succeeded. The problem with this type of approach is twofold. First, there is no way to indicate what kind of error occurred or even where in the method the error occurred. What if two or three primary operations were performed by the method? There is no way to tell which operations succeeded and which failed.

The second problem with this approach is the fact that, in many instances, meaningful data is passed back to the calling routine by the method. Suppose that we define a method called **doSomething()** to return a reference to an object of type **String**. Upon reaching an error, this method would return **null** instead of a String reference as an indication of an error. Take a look at what would happen in the following call, for example:

```
myObject. doSomething ( . concat ("World" );
```

The problem with this code is that any error that occurs in **doSomething()** would cause the

program to crash or come to an abrupt halt. The return value from **doSomething()** is being cascaded into the **concat()** call from the expected **String** object being returned. If the method returns **null**, what will the **concat()** method work on? **null** is not a **String** object. It has no methods to be called.

In some cases, however, it might be appropriate to use the return value of a method to indicate an error condition. When using the method, the user must be clear as to the meaning of the return value. We might want

to use a return value for error reporting if we are using a method such as the **indexOf()** method of the **String** class, for example. **indexOf()** is used to find the place in a string where the first instance of a character exists. The return value is that position. If the character is not found, the method returns -1. Why not throw an exception instead? An error condition is not always an exception. Remember that an exception is a condition in which program or data stability is suspect or actually corrupt. Not finding a character in a string doesn't pose any kind of impending threat to the program or data state. Although the method fails in its objective, the data still retains its integrity.

In cases like this, throwing an exception might cause more harm than good. Programming and runtime overhead are involved in exception handling. The cost of implementing the exception must be weighed against the benefits derived. With this in mind, continue to the next section, which takes a closer look at the mechanics of throwing and catching exceptions.

Throwing and Catching Exceptions

Any method in Java may throw any exception. In order to throw the exception, though, the method must be defined to throw it.[1] There is a **throws** clause that can be added to any method declaration to indicate that the method can throw an exception. We can list as many exceptions as appropriate for the method, as shown in this example:

```
Public void Foo()
      throws BarException, Bar2Exception
{
      . . .
}
```

To throw an exception (assuming that a **throws** clause exists that allows it), we need to create a new object of the type of the required exception class and use the keyword **throw** to deliver it, as this statement shows:

---

[1]A group of exceptions called *runtime exceptions* is subclassed from the Java **RuntimeException** class. These special *exceptions* do not need to be declared to be thrown. Any method may throw any *of* these exceptions at any time. The **RuntimeException** class is used to indicate a runtime error condition, such as trying to access a method on an unallocated class object or trying to access an out-of-bounds index on an array.


```
throw new IOException("Bad file name");
```

Catching an exception is a little more involved. We use a combination of statement blocks to define the relevant test and response actions. To let the system know that we are testing for the exception condition, we enclose the relevant code in **`try/catch`** blocks, as this code shows:

```
try
{
        . . . // code in danger of exception condition
}
catch( MyExceptionClass e )
{
        . . . // handler code
}
```

Here, a block of code follows the **`try`** statement. This is the code for which the exception will be tested. A second block of code also follows the **`try`** block; this is the **`catch`** block. The **`catch`** keyword always is followed by a declaration much like a method declaration with a single parameter. The parameter should be a derivative of the **`Exception`** class, and it must be a class type that has **`Throwable`** in its class hierarchy. This **`catch`** block is the actual exception handler.

If, in the process of executing the code in the **`try`** block, an exception is generated that is assignable to the exception class declared in the **`catch`** statement, the exception is said to have been *caught* At this point, the code in the **`catch`** block is executed in the thread in which the exception occurred. This code is treated much like a method call, but its scope is that of the method enclosing the **`try/catch`** blocks.

To perform exception handling, both blocks are necessary, complete with parentheses. Although it generally isn't a good idea to do this, either block can legally be empty. If the **`try`** block is empty, the entire code segment is a null operation. Obviously, if no code exists to test for the exception, no exceptions are caught.

The only reason to have an empty **`catch`** block is to prevent an exception from being propagated to the parent class. We must be certain that this is really what we intend before we do something like this, because it can have serious consequences in the running Java process.

In the preceding code example, there can be as much code as necessary in the **`try`** block. We are not limited to just the method call that may throw the exception. It is a good idea to keep the code to a minimum, though. Just include enough code to properly encapsulate the significant operations. If a repeated operation, such as a **`while`** loop, encloses a

method call, it might be appropriate to include the entire **`while`** loop in the **`try`** block instead of placing the **`try`** block within the **`while`** loop. This approach circumvents the repeated overhead of the **`try`** clause. Be sure that the **`catch`** block takes this fact into consideration, though, if the exception is not terminal.

A third statement block can be used optionally to enhance the functionality of the **`try/catch`** blocks: the **`finally`** block. The **`finally`** block offers a way to provide for the execution of a block of code after a **`try/catch`** block, whether or not an exception is caught. This clause

overrides any control-transfer statements invoked in the **`catch`** block, including any **`break`**, **`continue`**, or **`return`** statements as well as the propagation of the exception itself

The **`finally`** block of code is executed whether or not the exception is caught and whether or not an exception is even thrown. In terms of program flow, if the exception is not thrown, the **`finally`** block is executed immediately after the **`try`** clause completes. If the exception is thrown and is caught by the **`catch`** block, the **`finally`** block is executed immediately after the **`catch`** block but before any **`return`**, **`break`**, or **`continue`** statements. The **`finally`** block is used to ensure that our follow-up code always gets executed.

Suppose that we have a method that opens a file, performs some input and output operations, and then closes the file. If an exception condition occurs during the course of the input or output operations, we probably

Figure 2-1
IOTest.java

```
import java.io.*;

public class IOTest
{
    public static void writeFile( String name, String
          contents )
    {
        FileOutputStream f;
        try
        {
            f = new FileOutputStream( name );
        }
        catch( Exception e )
        {
            System.err.println( "Exception opening
            file "
                 + name +":" + e );
            return;
        }
```

Continues

Figure 2-1
Continued

```
        DataOutputStream out = new DataOutputStream(f);
        try
        {
            out.writeBytes( contents );
        }
        catch( IOException e )
        {
            System.err.println(
                "Exception writing to file:" + e );
            return;
        }
```

```
        finally
        {
            try
            {
                f.close();
            }
            catch( IOException e )
            {
                System.err.println(
                    "Exception closing file:" + e );
                return;
            }
        }
    }
    public static void main( String args[] )
    {
        writeFile("Test", "This is it, my friend" );
    }
}
```

will want to make sure that the file still closes. The **`finally`** block is perfect for this type of situation. Figure 2-1 shows a small test application to demonstrate this.

The **`writeFile()`** method performs three basic operations. It creates a new file named **`name`**, it writes **`contents`** to the file, and then it closes the file. Don't be daunted by the fact that this method has more than 30 lines of code; it is really quite simple and elegant. Each of the three operations may fail for reasons beyond the control of the programmer. There may be no room on the local file system, for example, or the user may not have permission to create a file. A number of conditions could cause the failure, so each operation is protected by a **`try/catch`** block. The **`writeBytes()`** method is protected by a **`try/catch/finally`** block to ensure that the **`close`** operation is executed. Note that the **`close()`** method is called after the catch's **`print`** statement but before the **`writeFile()`** method returns.

One final note on catching exceptions: Because all exceptions are subclasses of the **`Exception`** class, it is perfectly legal to define a **`catch`** statement such as this:

```
try
{
    . . .
}
catch( Exception e )
{
    . . .
}
```

Although this is perfectly legal, it generally is a very bad idea. The big problem with this example is that this **`catch`** statement will catch *any* exception. One of the great features of Java exception handling is that the **`catch`** statement only catches the exceptions it is defined to catch. Suppose that this **`try/catch`** block is protecting against an **`IOException`**. The **`catch`** statement as defined will catch the **`IOException`** if it occurs. The **`catch`** block then can handle the exception by whatever means has been defined. But what will happen if a

NullPointerException is generated? The **catch** statement defined here will catch this exception as well. This statement is designed to handle input/output exceptions, and yet it will be called for a **NullPointerException**.

If the **catch** block handles the exception simply by closing the stream and returning, the **NullPointerException** remains unhandled. This could lead to unforeseen problems with the rest of the application. It is almost always a bad idea to catch the **Exception** class instead of one of its subclasses.

In certain situations catching a more general exception can be useful. We could define an exception class that is the superclass to all the custom exceptions that can be thrown by classes in our project. We then could catch our base exception and differentiate it, if necessary, by using the **instanceof** operator in the **catch** block. But keep in mind that we must be very careful that we handle every exception in the group properly.

The Throwable Class

The base class for all exceptions in Java is the **Throwable** class. For an object to be thrown or caught, it must be derived from **Throwable** or one of the subclasses of **Throwable**. Two types of general **Throwable** objects exist: the Exception class and the **Error** class.

The **Exception** class is a special class, because the compiler enforces its throwing and catching. Unless the exception is generated by the Java virtual machine, the method doing the throwing must declare that it may throw the exception. Likewise, if a method declared to throw an exception is used, the exception must be dealt with in some fashion. We can deal with an exception in two ways: catch it or rethrow it. A class may define that it throws an exception. By declaring this, it alleviates the class's methods from explicitly handling that particular exception. In this case, the unhandled exception simply is propagated to the class that invoked the offending method. The **Exception** class itself doesn't add any user functionality to **Throwable;** it is just used as a superclass to other enforced exceptions.

The difference between the **Exception** class and the **Error** class is that **Error** is not bound by the compiler to be declared as being thrown or caught. These exceptions represent conditions that may occur during runtime that affect the virtual machine. **Error** generally is not intended to be caught by the program. It indicates a condition that in theory should not occur in a running program. An example of an **Error** subclass is the **StackOverflowError.** This error is thrown when the program stack in the Java virtual machine overflows. Errors are abnormal and generally unrecoverable; therefore, they are best left to the system to handle.

The exception model is defined this way to give all exceptions a common base that is not equivalent to **Throwable** and to differentiate **Exceptions** from**Errors.** Using this kind of no-op subclassing[2] ensures that, during type checking, all **Exceptions** are **Throwable**, but all **Throwables** are not necessarily **Exceptions.** Because the difference between the definitions of **Throwable** and **Exception** really is in name only, our exploration of the **Exception** class is also an examination of the **Throwable** class.

Exception defines only two constructors to override the **Throwable** constructors. These constructors are the only methods in the **Exception** class. Their sole functionality is to call

the corresponding **Throwable** constructor. The constructor takes no arguments at all or a single **String** object that is used as a detail message to the **Throwable** object.

The rest of the methods examined in this chapter are part of the **Throwable** superclass.

The method **getMessage()** returns the **String** object containing the detailed message. This may be **null** if no detailed message was supplied.

---

[2]We are referring here to creating subclasses without adding any new member fields or methods.

A standard **toString()** method exists, as in most methods (those methods that don't supply a **toString()** inherit the default from the **Object** class).

The method **fillInStackTrace()** populates the internal handle **backtrace** with the call stack information. The code that generates this information is native to the local platform. The format of this stack information is unspecified and also is platform dependent.

There are a couple of **printStackTrace()** methods that will each call the private native **printStackTrace0()** method, which will print the stack trace to either the **System.out** stream or to a **PrintStream** supplied by the caller.

Using the Built-in Exceptions

Now take a look at some of the most commonly used exceptions. The first **Exception** subclass we will explore is the **ArrayIndexOutOfBoundsException.** This exception is thrown when an attempt is made to access a member of an array with an invalid index. If the index supplied is greater than the length of the array less one or less than zero, it is invalid.

In this example, the index 3 is invalid:

```
int array [] = new int[3];
array[3] = 6;
```

This code would cause the **ArrayIndexOutOfBoundsException** to be thrown. The only valid indexes for **array** are 0, 1, and 2. This is one of the **RuntimeException** subclasses. Because this exception is generated by the Java virtual machine, it does not need to be declared explicitly as being thrown, and it is not mandatory that it is caught. If the exception is not caught, however, it leads to the invoking thread being shut down.

Another common exception in Java applications programming is the **IOException.** This exception is thrown when there is a problem with an input or output operation. Take a look at the following simple output operation, which creates a new file for output:

```
public FileOutputStream openOutFile( String name )
{
       try
       {
            return new FileOutputStream( name );
       }
```

```
        catch( java.lang.IOException e )
        {
              System.out.println("Unable to open file:" +
              name);
              return null;
        }
}
```

Suppose that an invalid file name is supplied to this method. The constructor for **`FileOutputStream`** declares that it throws **`IOException`**. Without the **`try/catch`** block or a declaration that the class will rethrow the exception, a compilation error will occur. This example assumes that the calling method will check the return value for **`null`** and handle the error. It could prompt the user for a new file name in the exception handler, for example.

Defining Our Own Exceptions

When designing our own exception classes, it is a good idea to follow the convention of keeping the exception class with the package from which it is intended to be thrown. The **`IOException`** class, for example, is part of the **`java.io`** package. This convention is a good idea for two reasons. First, it prevents the need to import a whole package of exceptions or to explicitly import each exception class into the class that uses it. If the code needs to catch an exception from an object's method, the package that defines the object (and therefore the package that defines the exception) already will have been imported. Second, the base **`Exception`** class is part of the **`java.lang`** package. It is not a good idea to make any modifications to the core packages, because that could lead to confusion when delivering the compiled classes or installing a new Java class library.

With that in mind, we will now create our own exception class. Because constructors have no return types, assume that we need to check that a constructor is initialized properly. To do this, we'll create a **`ConstructorException`** class. For this class, we want to include a way to determine the cause of the constructor failure. The base class provides a **`string`** object in which we can store detail information in the exception object. A lot of overhead exists in string manipulations, though. It is easier programatically to check a numeric value instead of the string. We will add an int field member to our class to store the numeric value. We also will define some constants that can be used to represent the different failure conditions. The class definition follows:


```
public class ConstructorException
      extends Exception
{
      public ConstructorException( String s, int cause )
      {
           super(s);
           this.cause = cause;
      }

      public ConstructorException( int cause )
      {
           this(null, cause);
```

```
        }

        public ConstructorException( String s )
        {
             this(s, UNKNOWN);
        }

        public ConstructorException()
        {
             this(null);
        }

        public int getCause()
        {
             return cause;
        }

        int cause;

        public static int UNKNOWN       = 0;
        public static int REASON_FOO = 1;
        public static int REASON_BAR = 2;
}
```

Notice that the **`ConstructorException`** class has two additional constructors in addition to the **`Exception`** superclass. We have allowed for the exception object to be created in as many ways as possible. The object requires two parameters: a string and an int. The user may create the exception using either, both, or none. Any parameters that are not passed as arguments are set to default values by the appropriate constructor. The string is set to null if it is not provided, and the int is set to the constant value **`UNKNOWN`** if it is omitted.

The string can be used to store meaningful text information in the event that the exception is rethrown all the way to the virtual machine level (through the propagation discussed earlier). If the exception gets that far without being handled, the executing thread halts, and the detailed message is printed along with a stack trace. The stack trace can be handy

for debugging purposes because it shows the call stack of all of the methods leading to the exception condition.

The int value can be used by a **`catch`** clause to determine the reason for failure and perhaps to allow recovery. If the cause of the failure was **`FOO`**, for example, perhaps something can be done to correct the situation, and then the constructor could be called again to instantiate the object.

That's really all there is to it. Defining our own exceptions is simply a matter of adding any exception-specific data to the base **`Exception`** class and supplying any constructors or accessor methods needed.

Exercises

1. Create individual classes that generate the following exceptions:

```
ArithmeticException
NullPointerException
ArrayIndexOutOfBoundsException
FileNotFoundException
ClassNotFoundException
```

2. Demonstrate the effects of catching and not catching (rethrowing) each type of exception from Exercise 1.

3. Develop a class called **`PositiveOnly`**. This class should take a positive integer value through its constructor. Use an instance method to decrement the value. Create a custom exception to be thrown whenever a negative value is reached. Use the exception handler to report this exception to **`System.out`** and set the value in the object to a non-negative value.

Summary

In this chapter, we learned the following:

- An *exception* is an unusual situation in which the state of the program is in jeopardy of becoming or has become unstable or corrupt.
- Return values can be used to indicate some error conditions upon the return from a method, but exceptions can give more detailed information about the cause of the method failure and can offer a better chance of error recovery in certain circumstances.
- Five Java keywords are used when throwing and catching exceptions: **`throws`**, **`throw`**, **`try`**, **`catch`**, and **`finally`**. We looked at using these keywords and deciding when it is appropriate to use each keyword to handle exceptions.
- The **`Throwable`** class is the base class for all **`Exception`** and **`Error`** classes. We briefly explored the methods in the **`Throwable`** class.
- How to use some of the built-in **`Exception`** classes.
- How to define customized **`Exception`** classes to be used in special situations.

Chapter 3
Arrays, Vectors, and Sorting

This chapter takes a brief look at the basics of array handling and explains the vector as a generic extensible array type. The treatment of arrays as objects in Java is discussed and examples are provided for the declaration and initialization of Java arrays. The reasons for

using vectors instead of arrays are outlined, and examples are given on how to extend vectors to provide functionality not available in a standard array. One of the examples in the chapter shows us how to use a container class to extend the functionality of the basic Java vector. The Quick Sort algorithm is explained, and a simple implementation is presented. The exercises near the end of this chapter include the development of a **`Sortable`** interface to create the **`SortedVector`** data type, which brings together these concepts.

What Are Arrays?

An *array* is a collection of data, all the same type, stored in contiguous memory. In Java, an array may be an array of primitive types, such as ints, floats, or chars. An array also can consist of reference types, including objects of the core classes and objects of a user-defined type. Arrays have a static number of elements set when the array is instantiated. After the array is created, the number of elements it can contain is static. Although all the elements of the array may not be populated with valid objects at any given moment, the size of the storage set aside for the array is fixed. Array variables (references) are reusable; to change the length of an array, we can create a new array of the desired length and assign it to the original array variable.

Arrays are a special data type in the Java language. Certain properties are special to arrays. In Chapter 1, "Basic Concepts," reference data types were discussed. Arrays are treated as reference types by the system, regardless of the type contained in the array. But at the same time, arrays are not classes as are other reference types. Also, arrays do not extend from the root class **`Object`**, although **`Object`** is considered the superclass of all arrays,[1] and an array can be treated as if it were of type **`Object`**. An array can be assigned to any variable of type **`Object`**. Any of the methods from **`Object`** can be called through an array. The memory for an array must be allocated using the **`new`** operator just like a class object, no matter what type is contained by the array.

```
int arrayOfInt[] = new int[7];
String arrayOfString[] = new String[4];
```

In the preceding declarations, **`arrayOfInt`** is an array of seven int values. Because int is a primitive type, there is no need to allocate any additional space after the new call. The array already is assigned enough contiguous memory to hold seven ints. The **`arrayOfString`** allocation call is a little different. The **`new`** operator assigns enough contiguous memory for the handles to four **`Strings`**; it does not allocate the memory for the **`String`** objects themselves. The **`String`** memory must be created separately, as shown in this code:

---

[1]See section 10.8 of the Java Language Specification (Gosling, Joy, Steele)


```
int arrayOfInt[] = new int[7];
String arrayOfString[] = new String[4];

for( int i = 0; i < 7; i++ )
{
```

```
        arrayOfInt[i] = i;
    }

    arrayOfString[0] = new String("Hello");
    arrayOfString[1] = new String("World");
    arrayOfString[2] = new String("It's");
    arrayOfString[3] = "Me";
```

The final statement is an implicit call to the following:

```
    arrayOfString[3] = new String("Me");
```

Access to the elements in an array is provided by the index operator ([ ]), as in the preceding example. All Java arrays use a zero-based index. This means that for an array of *N* elements, the valid index values are 0 to *N*-1. An array also has a special public instance member called length. The **`length`** member contains the allocated **`length`** of the array. This value is associated with the array object at allocation time and is not changeable during the life of the object.

What Are Vectors?

A *vector* is a type-safe, dynamic collection class similar to an array with advanced data-handling features. In a vector, the size of the collection is dynamic. Storage space can be added or deleted on-the-fly. This allows for efficient memory management on a data set that can vary in size. The vector also allows the addition, insertion, and deletion of data.

The vector class has three constructors, all of which are public. One of the protected member fields is **`capacityIncrement`**. This member keeps track of how much to grow the collection when memory needs to be allocated. The parameters to the constructors offer different levels of control over the initial size and **`capacityIncrement`**. The constructors for the **`vector`** class are as follows:

```
    public Vector(int initialCapacity, int capacityIncrement)
    public Vector(int initialCapacity)
    public Vector()
```


The first constructor enables the user to set both the initial size of the collection and the **`capacityIncrement`**. The collection size is analogous to the **`length`** member in an array. The second constructor sets the initial size but leaves the **`capacityIncrement`** set as the default. The third constructor sets both the initial size and the **`capacityIncrement`** to the defaults.

The **`vector`** class default size is 10. The default **`capacityIncrement`** is not a specific number; instead, it is an algorithm. If no specific capacity increment exists, the vector doubles the size of the collection every time it runs out of space. This process might seem inefficient, but it isn't. In most cases, it is very effective as a trade-off between speed and space management. Now take a closer look at how the vector manages memory.

The vector uses an internal array variable to store the data. As the array runs out of space, a new array is allocated based on the **`capacityIncrement`**. The data from the old array is copied to the new array, and the new array is assigned to the internal array variable. The

creation of the new array and the copying are relatively expensive in terms of time. To get the best possible performance out of a construct such as the vector, you need to minimize the number of expansions made to the collection. At the same time, to keep resource use to a minimum, you need to keep the unused storage space in the collection as small as feasible.

If you have a good handle on the data requirements, you can manage the growth of the collection programmatically. In many cases, though, you will not be able to accurately estimate the appropriate capacity and increment of the collection. In these cases, the default **`capacityIncrement`** can be very efficient; it is a good trade-off between memory and speed.

Consider the following scenario. A vector is created with an initial size of 1, and then 100 strings are added to the vector, one by one. Compare the capacity changes in Table 3.1 for the default increment versus an increment of 10.

Using the default **`capacityIncrement`** causes a resize only seven times, whereas using an increment of 10 requires 10 resizes. Here, the trade-off is three less array creations and copies against the unused space associated with the 28 extra elements that are allocated.

```
public final synchronized void copyInto(Object anArray[])
```

The **`copyInto()`** method takes an array as an argument and copies the entire contents of the vector into it, as shown in the preceding code. The collection must be of sufficient size to hold all the elements in the vector.

```
public final synchronized void trimToSize()
```


Table 3.1 Vector Capacity Changes

Capacity Change Iteration	Default Increment	Increment of 10
0	1	1
1	2	11
2	4	21
3	8	31
4	16	41
5	32	51
6	64	61
7	128	71
8		81
9		91
10		101

The **`trimToSize()`** method reduces the capacity of the vector to equal the number of elements contained, as shown in the preceding code. In this scenario, a call to

`trimToSize()` after string 100 is added reduces memory use from 128 strings to 100. This, of course, involves another iteration of the capacity change. This time, the capacity is reduced rather than increased.

```
public final synchronized
      void ensureCapacity(int minCapacity)
```

The preceding method checks the length of the internal array against the **`minCapacity`** argument. If the array is of greater or equal length, the method simply returns. If the array is shorter than the requested capacity, the vector is resized to the **`minCapacity`** supplied or to the next **`capacityIncrement`** step, whichever is greater

```
public final synchronized void setSize(int newSize)
```

The **`setSize()`** method enables the user to have explicit control over the size of the internal. Here, the array is set to the specified size, and the data in the original array that was beyond the new end of the array is truncated.


```
public final int capacity()
```

This method returns the current capacity of the vector.

```
public final int size()
```

This method returns the current number of used elements in the vector.

```
public final boolean isEmpty()
```

This method returns **`true`** or **`false`** to indicate whether the vector has any elements. This code is regardless of capacity; it refers strictly to the used elements.

```
public final synchronized Enumeration elements()
```

The **`elements()`** method returns an enumeration of the elements in the vector. Here, **`Enumeration`** is an interface that allows a single-pass walk-through of a data set. The **`Vector`** class provides its own specialized **`Enumeration`** class called **`VectorEnumeration`**. The methods are the standards provided by the interface declaration; no new methods are defined in **`VectorEnumeration`**.

```
public final boolean contains(Object elem)
```

The preceding method returns **`true`** or **`false`** to indicate whether the supplied object is contained in the array. Here, the object is compared by using the object's **`equals()`** method.

```
public final int indexOf(Object elem)
```

Using the same criteria as earlier, **`indexOf()`** returns the index of the desired element.

```
public final synchronized
      int indexOf(Object elem, int index)
```

The preceding method differs from the single-parameter version only in the fact that the search for the object starts at the supplied index instead of zero.

```
public final int lastIndexOf(Object elem)
```

This method performs the **`indexOf()`** search backward from the last element in the collection.


```
public final synchronized int lastIndexOf(Object elem, int
                                          index)
```

The preceding method is the same, except that the backward search begins at the specified index.

```
public final synchronized Object elementAt(int index)
```

The **`elementAt()`** method is an accessor method that provides the same index reference functionality as the index operators ([ ]) in an array. The element at the indicated index is returned. Here, the index must be valid for the current collection, or an exception is thrown.

```
public final synchronized Object firstElement()
```

This method returns the element at the zero index.

```
public final synchronized Object lastElement()
```

This method returns the last element in the collection.

```
public final synchronized
      void setElementAt(Object obj, int index)
```

This method enables the user to substitute a new object for the object contained in the element indicated. Here, the index must be valid for the current collection, or an exception is thrown.

```
public final synchronized void removeElementAt(int index)
```

This method enables the user to delete an element from the collection at the specified index. Here, the index must be valid for the current collection, or an exception is thrown.

```
public final synchronized
      void insertElementAt(Object obj, int index)
```

This method enables the user to insert an element at the specified index. Here, the index can be anywhere from zero to the number of elements. This enables the **`insertElementAt()`** method to be used also as an append operation (the last element in the list is indexed at one less than the number of elements-N-l). The list is checked for available space and expanded as necessary according to the **`incrementCapacity`** setting**.**

```
public final synchronized void addElement(Object obj)
```


The **`addElement()`** method appends the new object to the end of the collection. The list is checked for available space and expanded as necessary.

```
public final synchronized boolean removeElement(Object obj)
```

The preceding method removes the element from the collection that matches the specified

object. Here, the **`indexOf()`** method is used to find the first occurrence of the object in the collection, and that element is removed using the **`removeElementAt()`** method.

```
public final synchronized void removeAllElements()
```

This method empties the collection of elements. Each element, in turn, is set to **`null`**, and the element count is reset to zero.

```
public synchronized Object clone()
```

The **`clone`** method creates a copy of the **`vector`** object, not the elements. Here, the internal array is copied to a new **`Vector`** object, which then is returned. The vector is defined to implement the clonable interface; therefore, this method must be supported.

```
public final synchronized String toString()
```

This method generates and returns a string representation of the **`Vector`** object.

Only three instance variables are defined in the **`Vector`** class:

```
protected Object elementData[];
protected int elementCount;
protected int capacityIncrement;
```

All three variables are declared to be protected so that they are available to subclasses but not the general public. **`elementData`** is a generic **`Object`** array handle that holds the internal array storage for the collection. The **`elementCount`** member field is self-explanatory, and **`capacityIncrement`** was covered earlier in this chapter.

Vectors Versus Arrays

In many cases, a vector and an array may be used interchangeably. Generally, though, one is preferred over the other for any given circumstance. As a good general rule of thumb, we would use an array anytime the collection meets the following conditions:

- All elements of the collection are of the same type (especially the primitive types).
- The collection is a known, fixed size or maximum size.
- The collection is a non-sorted data set (data is not inserted into the collection).

If any of these conditions are not met, it might be better to use a vector object. Generally, the choice is pretty clear. If it necessary to manipulate the *storage* of the data, regardless of whether the data itself is manipulated, it probably is better to use a vector instead of an array. The advantages of the vector are mainly the fact that the collection is of a dynamic size and that methods are available to manipulate the storage.

If we want to perform an insert into an array, assuming that we have space for the additional member, we would have to do something along these lines:

```
System.arrayCopy( myArray, insertPosition,
      myArray, insertPosition + 1,
      array.length - insertPosition );
```

```
myArray[insertPosition] = newElement;
```

This is basically what the vector's **`insertElementAt()`** method does.

The advantages of using an array are speed, easy access, and reduced overhead. As mentioned earlier, the vector's capability to expand as necessary comes with the price of creating and copying the collection to the new internal array. By allocating all the memory needed for the array at one time, these time-consuming operations are eliminated. Each access to an element in a vector also requires a method call, which is avoided by the simple index notation an array uses for element access.

The advantages of a vector are flexibility and control. The methods available in the vector enable the programmer to manipulate the collection at will. It is a simple matter to insert or move an element in the collection.

Extending the Vector

In the Java **`Vector`** class, most of the methods are defined as final; this prohibits us from modifying the behavior of the class by preventing us from overriding the methods. This restriction also speeds up the performance of the **`Vector`** class. In Java, it is possible to have more than one method in the same class with the same name. The methods must have different argument lists, but they can all share the same name. This fea-

ture is known as *method overloading*. Methods inherited from a superclass may be overloaded as well. A subclass also may contain a method with a name and signature that are identical to a method in the superclass. This feature is known as *method overriding*.

When a class has methods that are overridden, there is a question of from which class (superclass or subclass) to actually invoke the method. The method called is the closest match to the choices available based on the runtime type of the object calling the method. It sometimes is impossible to determine at compile time which class an object really is. For this reason, the determination of which method to call is deferred until runtime. At runtime, the system can determine the exact real type of all the objects. Remember that an object carries not only its real type but also all the types of all its superclasses all the way down to the **`Object`** class itself

Any method that has the possibility of being overridden is called a *virtual method* All the virtual methods are kept in a table by the system so that the runtime environment can perform lookups to determine the correct virtual method to call. In Java, all methods are virtual by default. Methods declared as final, however, cannot be overridden by a subclass. This eliminates the need for a virtual function table and the overhead involved with looking up the appropriate method each time a call is made. Access to any final or private methods is faster than access to the virtual methods in the class.

The vector's instance variables are declared as **`protected`**, though. Protected access to the variables enables us to extend the functionality of the vector by allowing the subclass access to the internal data representation of the class. So, in effect, even though the core functionality of the class cannot be changed because of the final methods, the class may be extended at will by

adding functionality to the class.

Any kind of functionality can be added to the class as long as we don't need to change the behavior of any of the existing methods. In the next section, we'll take a look at extending the vector by adding the capability to sort the elements in the collection.

Creating a Sorted Vector

One feature that would be handy to have in a **`vector`** class would be the capability to have the vector automatically sort the data. Of course, in order to sort the data, we first must be able to compare one element to another. To accommodate this requirement, we will define a Java interface

called **`Comparable`**. The base interface needs to declare only one method: **`compare().`** We can define the **`Comparable`** interface by using the following code:

```
public inteface Comparable
{
      public int compare( Comparable obj );
}
```

This simple code segment says a lot. Remember that, in Java, an interface does not implement any methods; it just declares them. Any class that implements the interface will implement the methods declared. By implementing the **`Comparable`** interface, an instance of the class becomes an object of type **`Comparable`** as well as its other inherited types. This lets us use the **`instanceof`** operator from within our sortable **`Vector`** class to confirm that an object is comparable before we attempt to call the **`compare()`** method when sorting the object within the vector.

The **`compare()`** method should perform an operation similar to the **`compareTo()`** method in the **`String`** class. The method should compare itself to the object passed as an argument and return an integer comparison value. The return value should be zero if the two objects are considered equal, negative if this object is less than the argument object, and positive if this object is greater than the argument object.

So, after we define the **`SortedVector`** class to use the **`compare()`** method to determine sort order, any class that implements **`Comparable`** can be stored in sorted order. But what happens if an object that is not comparable is passed to the sortable vector? We can handle this situation in one of two ways. We can supply a default order for non-comparable objects, or we can throw an exception indicating this as an unacceptable condition.

Another condition to consider is the case in which two objects implement the **`Comparable`** interface and yet are of different types. Suppose that the user defines a **`ComparableFoo`** and a **`ComparableBar`** class and tries to store both in the **`SortedVector.`** How is this situation handled? It is irrelevant to the **`Sortedvector`** class what type of objects it stores, as long as they implement the **`Comparable`** interface. The responsibility is on the user of the class to be sure that any comparable objects inserted into the collection know how to compare themselves to other comparable objects.

One of the benefits of using an interface to define a data type is that it enables classes like **`SortedVector`** to be used very flexibly. If the user can find some way to compare two different classes, both classes may be stored in the **`SortedVector`** together, as long as they both implement the **`Comparable`** interface.

Even though we can't override the vector's methods, there is nothing stopping us from overloading the methods in the vector. *Overloading* is defining methods with the same names that use different parameter lists. So, if we define the **`addElement()`** method to take a comparable instead of an object, we allow the user to add comparables to the list using the method defined in **`SortedVector`** instead of the superclass' **`addElement(Object).`**

Using this kind of subclassing requires good documentation. There is no way to stop the user from inadvertently adding an object that is not a comparable. If the user calls **`addElement()`** with a **`String`** argument, the vector's **`addElement()`** processes the call and destroys the sort order of the collection. However, as long as the user adds only comparables to the collection, the methods in **`Sortedvector`** add them in appropriate order.

Figure 3-1 shows the complete implementation of adt.Chapter03.SortedVector.

Figure 3-1
SortedVector.java.

```
package adt.Chapter03;

import java.util.Vector;
import java.util.Enumeration;

public class SortedVector
      extends Vector
{
    public SortedVector( int initialSize, int capacityIncrement )
    {
        super( initialSize, capacityIncrement );
    }

    public SortedVector( int capacityIncrement )
    {
        super( capacityIncrement );
    }

    public SortedVector()
    {
        super();
    }

    public void addElement( Comparable o )
    throws SortableException
    {
        int index;

        index = getInsertIndex(o);
```

Continues

Figure 3-1
Continued

```
            insertElementAt(o,index);
        }

        public void insert( Comparable o )
            throws SortableException
        {
            addElement(o);
        }

        public boolean contains( Comparable o )
        {
            return super.contains(o);
        }

        public SortedEnumeration sortedElements()
        {
            return new SortedEnumeration(elementData);
        }

        private int getInsertIndex( Comparable o )
            throws SortableException
        {
            int index;
            for( index = 0; index < elementCount; index++ )
            {
                if( !(elementData[index] instanceof
                       Comparable) )
                    throw new SortableException( "Element
                    " + index + " not Comparable" );

                if( o.compare( (Comparable)elementData
                                [index] ) < 0 )
                    break;
            }
            return index;
        }
    }

    class SortedEnumeration
        implements Enumeration
    {
        SortedEnumeration( Object array[] )
        {
            this.array = array;
        }

        public boolean hasMoreElements()
        {
            if( index < array.length )
```

```
            return true;
        return false;
    }
```

Continues

Figure 3-1
Continued

```
    public Object nextElement()
    {
        if( !(array[index] instanceof Comparable) )
            throw new SortableException(
                "Element " + index
                + " not Comparable" );

        return array[index++];
    }

    Object array[];
    int index = 0;
}
```

A **`SortableException`** class also is defined to handle any objects inserted into the collection that are not comparable. Although there is no way to stop this from happening because the vector itself will store any object, we can test that each element encountered in the **`addElement()`** and **`insertElement()`** is comparable as we insert each new object to determine that it is a **`Comparable`.** Each element also is tested as it is passed through the **`sortedElements()`** method. If it is not an instance of **`Comparable`,** the exception is thrown. These two checks ensure that none of the methods in the **`Sortedvector`** class will generate a runtime error because of data of the wrong type being included in the collection.

We now can create a small application to demonstrate the **`SortedVector`;** this application is called **`SortableVector`**. To use the **`SortedVector`,** we need to define a data class to implement the **`Comparable`** interface. The data class is called **`ComparableString`,** as shown in Figure 3-2.

To keep the example simple, the **`ComparableString`** class is not well encapsulated. The **`String`** member should be declared as **`private`,** and it should really have accessor methods. The important part of the class is the **`compare`** method, because it is the implementation of the **`Comparable`** interface. Because this is a comparable **`String`** class, it uses the built-in **`compareTo()`** method in the **`String`** class and saves the work of reimplementing the functionality.

The **`Sortablevector`** application simply declares and instantiates a **`SortedVector`** object and fills it with **`ComparableString`** objects (see Figure 3-3). Notice that all the **`addElement()`** calls are enclosed in a single **`try`** block. If we inadvertently try to add an object that is not comparable, the **`SortableException`** is thrown. After all the **`ComparableString`**'s are added to the collection, the application demonstrates that they are sorted by walking through the **`enum`** enumeration.

Figure 3-2
ComparableString. java.

```
package adt.Chapter03;

public class ComparableString
      implements Comparable
{
    public ComparableString( String s )
    {
        string = s;
    }

    public int compare( Comparable obj )
    {
        return string.compareTo(((ComparableString)obj).
                                  string);
    }

    public String string;
}
```

Figure 3-3
SortableVector.java.

```
package adt.Chapter03;

import java.util.Enumeration;

public class SortableVector
{
    public static void main( String arg[] )
        throws java.io.IOException
    {
        ComparableString cString;

        SortedVector v = new SortedVector();
        try
        {
        v.addElement( new ComparableString( "This" ) );
        v.addElement( new ComparableString( "is" ) );
        v.addElement( new ComparableString( "a" ) );
        v.addElement( new ComparableString( "sorted" ) );
);
        v.addElement( new ComparableString( "vector" ) );
);
        v.addElement( new ComparableString( "in" ) );
        v.addElement( new ComparableString( "sort" ) );
        v.addElement( new ComparableString( "order" ) );
        }
        catch( SortableException e )
        {
            System.out.println(e);
```

Continues

Figure 3-3
Continued

```
            System.exit(0);
        }

        Enumeration enum = v.elements();
        while( enum.hasMoreElements() )
        {
            System.out.println(
((ComparableString)enum.nextElement()).string );
        }
        System.in.read();
    }
}
```

Figure 3-4
The output from the data set {"This", "is", "a", "sorted", "vector", "in", "sort", "order"}

```
This
a
in
is
order
sort
sorted
```

Figure 3-4 shows the output for the application.

The **`SortedVector`** class maintains the sort order by managing the add and insertion operations. No special sort-in-place operation restores sort order if the order of elements in the collection changes. This type of utility class relies on the programmer to use the class properly. It is relatively easy to shoot yourself in the foot if you are not careful. This type of design enables the programmer to be much more flexible in implementing a solution, but it comes with a price. Other, somewhat safer, approaches to the sorted vector problem exist.

External Vector and Array Sorting

As an alternative to extending the **`Vector`** class, sorting can be done externally by a utility class. In this case, the data items are added to the collection normally, using the methods in the **`Vector`** class. At some point in the process, the collection is passed through a sort engine that reorders it appropriately. The sort engine could support any one of a number of sort algorithms that could be used interchangeably according to the user's preference. In this chapter, we will take a look at one of these types of generic sort algorithms: the quicksort.

The quicksort algorithm has a worst-case time of N^2, where *N* is the number of elements in the collection to be sorted. So, at most, N^2 comparisons must be made between the elements in the collection to completely sort the collection. This might not seem very efficient, but the average is closer to NlogN This can be proven mathematically, but such proof is beyond the scope of this book. Quicksort usually is considered to be a fairly efficient general-purpose sort algorithm. The general theory behind this algorithm is *divide and conquer*. The collection to be sorted is split into two subsets that are recursively put through the same process. The key to the algorithm is the process by which the subsets are determined. A boundary value is defined so that each element of subset A compares as less than the boundary value, and each element in subset B compares as greater than the boundary value.

Given an array **`A[]`** of length N (where LO=0, HI=N, and X=A[LO]), partition the array so that all elements less than X are to the left of X, and all values to the right of X are greater than X in the array. X can be any element in the array except for **`A[N-1]`**. If the boundary value is the last element in the array and it also is the lowest sort value in the collection, we will end up in an infinite recursion. The partitioning is accomplished by walking through the array in both directions and comparing the elements encountered to X. When an element in the ascending or descending leg of the walk compares as out of place, that leg is halted until the other leg is complete. After both legs are complete, the indexes of each leg are compared. If the ascending leg index is less than the descending leg index, the elements are swapped, and the process continues. If the ascending leg index is greater than the descending leg index, the array is partitioned at the descending leg index, and the two subarrays go through the same process.

Figure 3-5 shows an array of seven ints [6,3,8,5,7,1,9] in various stages of being sorted. The **`quicksort`** algorithm is used to sort the array.

- (A) shows the initial array. The value of the first element of the array, 6, is used as a pivot. [9] compares greater than 6, so the [9] is in the correct half and the arrow moves to the next node in line: [1]. [1] is less than the pivot value 6, so the arrow stops at [1]. The first element in the array, [6], compares (as expected) equal to the pivot value, so the arrow stops here. 6 is less than 1, so the two elements are swapped.

- (B) The comparisons continue after the swap with the top arrow at [7]. [7] is greater than 6, so the arrow moves to [5]. 5 is less than 6, so the arrow stops. The other arrow would have been pointing at [1] after the swap. That arrow moves to [3], which is less than 6. The

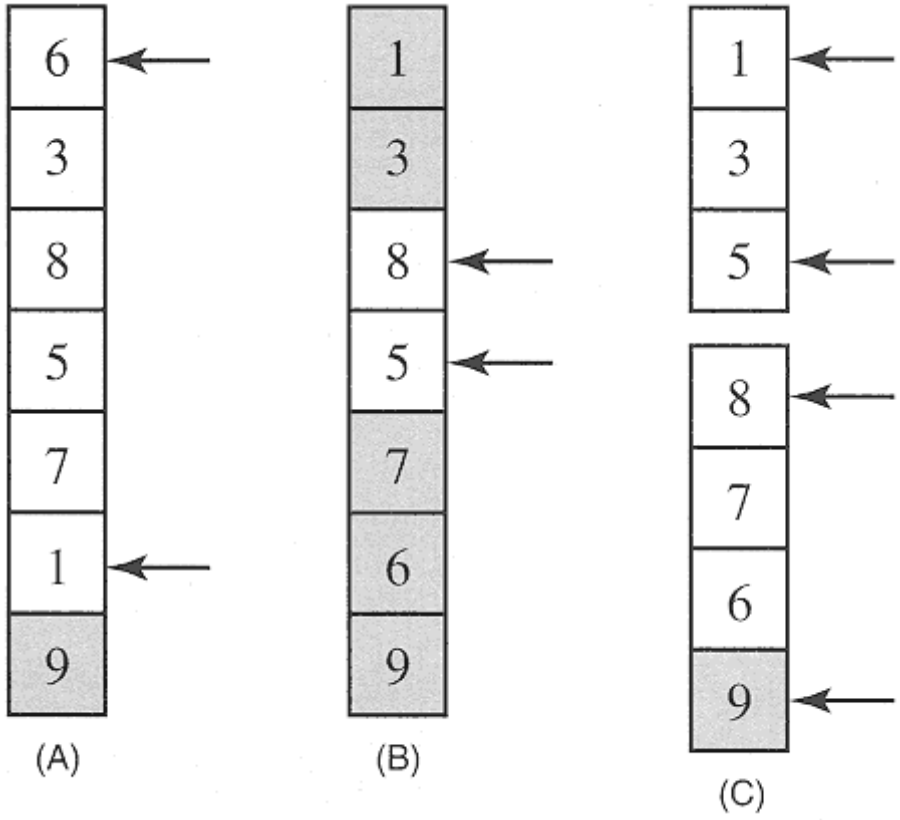


Figure 3-5
Three intermediate stages of a data set run through the quick sort algorithm.

arrow moves again to [8] which is, of course, greater than 6. Once again, the arrow stops. Because both arrows are stopped, a comparison is made between the elements at each arrow, and it is determined that a swap needs to take place between [8] and [5].

- (C) Because the arrows have met, we know that all the values below the split are less than 6 and all those above the split are greater than 6. Therefore, you can split the array into two subarrays at the arrows and start the process over again. Each subarray gets a new pivot value based on the first element in each subarray.

For the purpose of this example, we will define two classes that handle the sorting instead of subclassing the **`Vector`** class. The first class is the sort engine. It has a **`sort()`** method that can be called to sort an array or a vector. The engine class is called **`SortEngine`.** We still need a method of comparing the objects in the collection. For this, we'll define a new interface with a **`compare()`** method. The interface is called **`SortInterface`.**

Figure 3-6
SortEngine.java.

```
package adt.Chapter03;

import java.util.Vector;
```

```
public class SortEngine
{
      public SortEngine( SortInterface s )
      {
          helper = s;
      }

      public void sort( Object array[] )
      {
          quicksort( array, 0, array.length );
      }

      public void sort( Vector v )
      {
          quicksort( v, 0, v.size() );
      }

      public void quicksort( Object array[], int lo, int
                             hi )
      {
           if( lo == hi )
               return;

           Object o = array[lo];
           int i = lo - 1;
           int j = hi;

           while( true )
           {
               while( --j >= lo )
               {
                   if( helper.compare( array[j], o )
                       <= 0 )
                        break;
               }

               while( ++i < hi )
               {
                   if( helper.compare( array[i], o )
                       >= 0 )
                        break;
               }

               if( i < j )
               {
                   Object tmp = array[i];
```

Continues

Figure 3-6
Continued.

```
                array[i] = array[j];
                array[j] = tmp;
                j++;
```

```
            i--;
        }
        else
        {
            break;
        }
    }
    quicksort( array, lo, j );
    quicksort( array, j+1, hi );
}

public void quicksort( Vector v, int lo, int hi )
{
    if( lo == hi )
        return;

    Object o = v.elementAt(lo);
    int i = lo - 1;
    int j = hi;

    while( true )
    {
        while( --j >= lo )
        {
            if( helper.compare( v.elementAt(j),
                o ) <= 0 )
                 break;
        }

        while( ++i < hi )
        {
            if( helper.compare( v.elementAt(i),
                o ) >= 0 )
                 break;
        }

        if( i < j )
        {
            Object tmp = v.elementAt(i);
            v.setElementAt( v.elementAt(j), i);
            v.setElementAt(tmp,j);
            j++;
            i--;
        }
        else
        {
            break;
        }
    }
```

Continues

Figure 3-6
Continued.

```
        quicksort( v, lo, j );
        quicksort( v, j+1, hi );
    }

    public SortInterface helper;
    public final static int QUICK = 1;
}
```

<><><><><><><><><><><><><>

```
package adt.sort;

public interface SortInterface
{
      public int compare( Object a, Object b );
}
```

The difference between the **`Comparable`** interface and the **`SortInterface`** is that, in this case, the **`compare()`** method compares two separate objects. In the **`Comparable`** interface, a **`Comparable`** object compares another **`Comparable`** object to itself After we pass the sort engine an object that implements **`SortInterfac`**e, we can use the **`compare()`** method to compute the sort order.

To better understand the **`quicksort`** algorithm, we can examine the **`quicksort()`** method. The array version is a little easier to follow. The **`quicksort()`** method is called with three arguments: the array to be sorted and two integer values. The integer values represent the low and high index range of the array to be sorted. These values are necessary to have the capability to sort subarrays within a larger array. Initially, the method is called with the array, zero, and the array length, respectively.

In this implementation, the low index always is used to determine the partitioning boundary. The method initializes int's **`i`** and **`j`** to keep track of the current positions of each leg of the array traversal. The **`j`** leg compares each element of the array by descending index to the boundary value to determine whether the element belongs in the greater than subset. As soon as a value is found that is less than or equal to the boundary value, the **`j`**-leg is stopped, and the **`i`**-leg begins. The **`i`**-leg starts at the low end of the array and moves upward, comparing each value to the boundary value to determine whether it belongs to the less than subset. When a value is found that is greater than or equal to the boundary value, the **`i`**-leg is stopped.

If the **`i`**-leg stopped at a lower index than the **`j`**-leg, the elements are swapped, and the comparisons continue through the **`while`** loop. If the **`i`** and **`j`**-legs stop on the same index or if they cross, the array is parti-

tioned around the boundary value. In this case, the array is split around the boundary, and the two subsets are put recursively through the same process. After the size of any subset gets to 1, the element indexed is in the right place. So, after the recursion is done, the array is sorted!

Figure 3-7 shows a small console application to test the sort engine.

This example creates a vector with the words "This is a sorted vector in sort order" as its elements. It then creates a sort engine using **`this`** as the sort interface. The vector is sorted and displayed in order. The application then creates an array of strings containing "This is a sorted array in sort order," which it then sorts and displays. Notice that it does not really matter to **`SortEngine`** from where the **`SortInterface`** came. It could have been implemented in a separate class **`StringSortInterface`.** In this case, it was more convenient to implement it in the main class.

Figure 3-7
SortTest.java

```
package adt.Chapter03;

import java.util.Enumeration;
import java.util.Vector;

public class SortTest
    implements SortInterface
{
    public static void main( String arg[] )
        throws java.io.IOException
    {
        SortTest test = new SortTest();
        SortEngine engine = new SortEngine(test);

        Vector v = new Vector();

        v.addElement( "This" );
        v.addElement( "is" );
        v.addElement( "a" );
        v.addElement( "sorted" );
        v.addElement( "vector" );
        v.addElement( "in" );
        v.addElement( "sort" );
        v.addElement( "order" );

        engine.sort(v);

        Enumeration enum = v.elements();
        while( enum.hasMoreElements() )
        {
            System.out.println( (String)enum.
            nextElement() );
        }
```

Continues

Figure 3-7
Continued.

```
        String array[] = { "This","is","a","sorted",
            "array","in","sort","order" };
```

```
        engine.sort(array);
        for( int i = 0; i < array.length; i++ )
            System.out.println( array[i] );

        System.in.read();
    }

    public int compare( Object a, Object b )
    {
        return ((String)a).compareTo((String)b);
    }
}
```


Exercises

1. Create a small application to create, populate, and list an array of strings.

2. Create and use a vector to hold the same strings.

3. Modify the **`SortInterface`** to handle generic collections. This requires at least these additional methods: **`swap`**, **`getPrevious`**, and **`getNext`**. Then modify **`SortEngine`** and **`SortTest`** to use this new model to sort a vector and an array using the same call to **`SortEngine.sort()`**.

Summary

In this chapter, we learned the following:

- *Arrays* are collections of data stored in contiguous memory.
- A vector is a type-safe, dynamically sized implementation of an array.
- Sometimes, it is more appropriate to use an array, and sometimes a vector can be the best storage mechanism.
- The **`Vector`** class can be extended in various ways to be more useful.
- Sorting can be applied to the **`Vector`** class in several ways.

Chapter 4
Hash Tables

This chapter examines the hash table. A *hash table* is a container that enables quick-and-easy storage and retrieval of data that has a unique key associated with it. Hash codes and hash

methods are discussed in detail in this chapter. A simple hash table class is defined from scratch to demonstrate these concepts. We'll learn how and when to use hash tables, and we'll look at examples that use the core Java class **`Hashtable`** and its subclass, the **`Properties`** class. Near the end of this chapter, we'll run through some exercises on using the **`Properties`** object to parse command-line arguments.

What Are Hash Tables?

Hash algorithms describe a mapping of objects to a range of integer hash values so that the distribution of mapped objects is as even as possible. *Hash tables* are a collection of a fixed number of containers, with one container for each integer in the range of possible hash values. When an object is stored in a hash table, a hash value is computed, and the object is placed in the hash table's container that corresponds to the hash value.

A hash table is a storage container that associates a unique key with each object it stores. The key used to store the object is used later to retrieve the object from the table. Hash tables enable us to easily access data without having to linearly search through the entire set of data in the container. As the name implies, the data is stored in a table that is similar in many ways to a standard array. In fact, a hash table is really a generalized implementation of an array. Take a minute to examine what that means. In an array, data is stored in a tabular fashion. The data is accessed through an index into the array. If the index is known in advance, the data can be accessed with a single lookup operation such as this:

```
String myString = array[n];
```

In the case of the array, we can think of the index as a key. Arrays do require that the index be unique. Two data items cannot exist at the same index in the array. Arrays also require a static size. The array cannot be grown to larger than the initial capacity. So the array has a fixed set of keys (indexes) and a fixed storage capacity.

Because the hash table stores the data items in a tabular format, the hash table also requires that the *keys* are unique. One difference between an array and a hash table is that, in a hash table, the universe of keys generally is considered the set of natural numbers. The main difference between an array and a hash table is that, in the array, only one data item is stored in each index. In the hash table, the storage scheme is a bit different. The hash table generally has an internal array, but this array is an array of lists of key/value pairs. The indexes in the array represent the key lists in which to search for the specified key. It is desirable—but not necessary—to have a single object in each index.

A hash code is a numeric representation of an object. Suppose that we have a list of data items for which the domain of keys consists of two character strings (such as "aa", "ab", "ac", . . ., "ba", "bb", "bc", and so on). To translate the keys into numeric indexes that can be used in the array, a hash method must be defined.

The hash algorithm implemented must satisfy two basic requirements. First, it must guarantee that the same key always will produce the same hash code. The entire purpose of the hashing is

defeated if the hash codes aren't consistent. Second, the algorithm should have a reasonable chance of evenly distributing the expected universe of keys among all the possible hash codes. If the hash codes aren't evenly distributed, the efficiency of the hash table is degraded. Take a look at a simplified example of a hash algorithm. Suppose that, like the earlier example, the keys are all two-character strings, and all the characters are between 'a' and 'z'. The hash code generated is the ASCII value of the first character of the key minus 'a'. This generates a hash code between 0 and 25. The hash method could look like the one in Figure 4-1.

In Figure 4-1, the keys are hashed to the values of 0, 0, 0, 1, 1, and 1, respectively. Notice that the keys "aa", "ab", and "ac" are all hashed to the same value: 0. "ba", "bb", and "bc" are all hashed to the same value: 1. These conditions are known as *collisions* When keys collide, the key list at the appropriate index grows. The list of keys at any given index is commonly known as the *bucket.* All keys with a hash code of zero are put in the zero bucket, as shown in Figure 4-2.

If a user retrieves the data associated with "ba", the hash table lookup is simplified in comparison to a linear search construct, such as a vector. ba" hashes to 1, so bucket one is traversed linearly until the key is found. This reduces the maximum number of lookups to the number of keys in the bucket. In a vector, the entire set of keys would have to be traversed until the desired one is found, giving an average number of lookups of N/2 (N

Figure 4-1
A simple hash method.

```
public int hash( String key )
{
      byte array[] = new byte[1]
      key.getBytes ( 0, 1, array, 0 );
      return (int) (array [0]-(byte) 'a')
}
```

Figure 4-2
Multiple keys sharing the same "bucket" based on the hash algorithm provided.

Bucket	Key List
0	"aa", "ab", "ac", ... , "az"
1	"ba", "bb", "bc",...,"bz"
...	
25	"za", "zb",...,"zz"

is the number of keys in the data set). In this particular hash table example, the average number of lookups is 26/(N*2), assuming that there is an even distribution of keys. If this is the case, storing any list of more than five keys is more efficient (on average) in the hash table than in the vector.

This is a very crude and simplified example; its purpose is to demonstrate the basic concepts of hashing. The biggest problems with this example are that the table size is static and there is no allowance for optimization. What would happen if all the entries in the hash table had a key that started with 'a' or 'b'? The first two buckets would fill up, and the remaining 24 buckets would stay empty. This, in turn, would lead to inefficient linear lookups within the two filled buckets.

Also consider what could happen if the keys were three characters instead of two. With two-character keys, the table can hold a maximum of 676 entries, or 26 per bucket. If the key size is expanded to three, the table holds 17,576 entries. There are still only 26 buckets to hash into, with a new maximum of 676 keys per bucket. As we can see, the number of buckets and available keys can have a great effect on how efficient the hash table is. For the hash table to be useful in real-world applications, we'll need a better hashing algorithm. In the next section, we'll take a closer look at the hashing algorithm.

A Simple Hash Table

Up to this point in the chapter, we have examined the basic concepts of the hash table. Now take a look at a simple but fairly complete hash table construct. For the purpose of this demonstration, assume that all keys are unique **`String`** objects and that data may be any Java object derived from the **`Object`** class.

First take a look at the hash code generator method. It takes a string "key" as an argument and generates an int that represents the sum of all the characters in the string. Figure 4-3 defines the **`hashCode()`** method.

Now take a look at the entire class in Figure 4-4. The first thing we will notice is the definition of a non-public class called **`HashObject`**. The **`HashObject`** class is simply a data structure; it has no methods. This class is used for holding the key/value pairs and forming the basis of our list in each bucket.

In the **`HashTable`** class, the provided constructor takes no arguments. It simply initializes the bucket table to the correct starting size by creating an array of **`HashObjects`** to serve as the buckets in the table. The size

Figure 4-3
A simple *`hashCode`* method.

```
public int hashCode ( String key )
{
    int value = 0;

    byte array[] = new byte[key.length()];
    key.getBytes ( 0, key.length(), array, 0 );
    for ( int i = 0; i < array.length; i++ )
    {
    value += (int)array[i];
    }
```

```
    return value;
}
```

Figure 4-4
HashTable.java.

```
package adt.Chapter04;

class HashObject
{
    String key;
    Object data;
    HashObject next;
}

public class HashTable
{
    public HashTable()
    {
        table = new HashObject[23];
        size = table.length;
        rehashSize = 4;
        capacityIncrement = 2;
        count = 0;
    }

    public void put( String key, Object data )
    {
        HashObject obj = new HashObject();
        obj.key = key;
        obj.data = data;
        obj.next = null;

        bucketAdd( table, getBucket(hashCode(key)), obj
);
        count++;
        if( count > size * rehashSize )
            rehash();
    }
```

Continues

Figure 4-4
Continued.

```
    public Object get( String key )
        throws NoSuchKeyException
    {
        HashObject place = table[ getBucket
                                  (hashCode(key)) ];
        while( place.key.compareTo(key) != 0 && place
                                  != null )
            place = place.next;
        if( place == null )
            throw new NoSuchKeyException( key );
```

```
        return place.data;
    }

    private int hashCode( String key )
    {
        int value = 0;

        byte array[] = new byte[key.length()];
        key.getBytes( 0, key.length(), array, 0 );
        for( int i = 0; i < array.length; i++ )
        {
            value += (int)array[i];
        }

        return value;
    }

    private int getBucket( int hash )
    {
        return hash % size;
    }

    private void bucketAdd( HashObject table[],int
          bucket, HashObject obj )
    {
        obj.next = table[ bucket ];
        table[ bucket ] = obj;
    }

    private void rehash()
    {
        int newSize = size * capacityIncrement;
        HashObject newTable[];
        HashObject tmp, obj;

        if( newSize % 2 == 0 )
            newSize++;

        newTable = new HashObject[ newSize ];

        for( int i = 0; i < size; i++ )
        {
            tmp = table[i];
```

Continues

Figure 4-4
Continued.

```
            while( tmp != null )
            {
                obj = new HashObject();
                obj.key = tmp.key;
                obj.data = tmp.data;
```

```
                obj.next = null;
                bucketAdd( newTable, hashCode
                    (tmp.key) % newSize, obj );
                tmp = tmp.next;
            }
        }
        size = newSize;
        table = newTable;
    }

    private HashObject table[];
    private int size;
    private int rehashSize;
    private int capacityIncrement;
    private int count;
}
```

variables also are set to their default values. These variables are used to track the size of the table and to determine when to "rehash" the entire construct. It is important to use an odd number, preferably a prime, as the initial size of the table. This helps to optimize the distribution of keys in the table. We easily could add a constructor and accessor methods to enable the user to fine-tune the hash table performance, but for this example, we'll leave these out.

The only **`public`** methods defined beyond the constructor are **`put()`** and **`get()`.** The **`put()`** method takes a string **`key`** and an object **`data`** as parameters. A new **`HashObject`** instance is created to hold the key/data pair. The bucket into which this key belongs is determined by using the **`hashCode()`** and **`getBucket()`** methods. This is also the point at which the counter is bumped. If the total item count exceeds the maximum defined by **`size * rehashSize`,** the table is rehashed. Assuming that there is an even distribution of keys to the buckets, the hash table is rehashed when there are **`rehashSize`** entries in each bucket.

The **`get()`** method is used to retrieve the data by looking up the supplied key. The bucket number is determined by using the same process as the **`put()`** method. After the appropriate bucket is determined, the list at that bucket is traversed linearly until a matching key is found. If the key is not found in the list, an exception **(`NoSuchKeyException`)** is thrown to indicate the error condition. Figure 4-5 shows the definition for **`NoSuchKeyException`** It is nothing more than a no-op subclass of exception.

Figure 4-5
NoSuchKeyException.java.

```
package adt.Chapter04;

public class NoSuchKeyException
     extends Exception
{
     public NoSuchKeyException( String s )
     {
          super(s);
     }
```

```
    public NoSuchKeyException()
    {
        super();
    }
}
```

This kind of subclassing sometimes is used to indicate specific exceptions instead of the generalized Java core **`Exception`** class. This also makes it easier to add more specific behaviors or information to the exception at a later time.

Note that the private **`hashCode()`** method defined in this class is not really necessary in real-world Java applications. It is provided as an example of one specific method in which to generate hash code values. The core Java **`Object`** class, which is the parent to all Java classes, defines a generalized **`hashCode()`** method for use by all objects. This method returns a large int value for any instance of any object in the system. This hash code is not guaranteed to be unique, but it is guaranteed to always generate the same value for the same instance of any object.

Any particular class may override the default method to provide specific hashing functionality. The **`String`** class, for example, defines its own **`hashCode()`** method so that any two equivalent strings hash to the same value. The default **`hashCode()`** method provided by **`Object`** would generate different values for different instances of equivalent strings.

The remaining private methods are **`getBucket()`**, **`bucketAdd()`**, and **`rehash()`**. The **`getBucket()`** method calculates the bucket number by returning the modulus of the hash value provided as an argument and the total number of buckets as defined by **`size`**. This guarantees a bucket value between zero and **`size`** for each hash code.

The **`bucketAdd()`** method prepends the new **`HashObject`** to the list in the appropriate bucket. If there is no list in the bucket, the new **`HashObject`** is used to start the list. Otherwise, the first key in the list is assigned to the new **`HashObject`**'s **`next`** reference, and the new object becomes the beginning of the list. The table to which we are adding is supplied to the method to make this method useful to the **`rehash()`** method.

The **`rehash()`** method gives the hash table the capability to expand without corrupting the hash table scheme. Remember that the number of lookups required to find the key is directly proportional to the number of keys in any given bucket. If 100 keys are spread evenly among 4 buckets, the average number of lookups is 12.5. The same 100 keys spread evenly between 10 buckets yields a lookup average of 5. When the table gets too full, the number of buckets expands. To accomplish this, a new table must be created (because an array cannot change size), and all the keys must be reassigned to the new table in the correct new buckets. After the new table is fully populated, it replaces the old table, and the size variable is adjusted to reflect the larger table.

To test the example **`HashTable`** implementation, we can create a small Java application that uses this class. The application shown in Figure 4-6 loads 2000 key/value pairs into the hash table and then accesses them in the reverse order in which they were inserted.

Figure 4-6
HashTest.java.

```
package adt.Chapter04;

public class HashTest
{
    public static void main( String args[] )
    {
        HashTable table = new HashTable();

        for( int i = 0; i < 2000; i++ )
        {
            table.put("STRING " + i, new Integer( i )
                      );
        }

        for( int i = 1999; i >= 0; i-- )
        {
            try
            {
                System.out.println( "KEY = STRING " +
                                    i +
                    " VALUE = " + table.get("STRING " +
                                            i ) );
            }
            catch( NoSuchKeyException e )
            {
                System.out.println( "KEY NOT FOUND: "
                    + e.getMessage() );
            }
        }
    }
}
```


The only operation this sample hash table doesn't include is the capability to remove a key/value pair from the table. Other than that, it is a reasonably usable construct as it stands. The Java core classes, however, include a **`Hashtable`** class that is more robust than this example and includes several other useful features for real-world applications.

The Java Hash Table

The Java core class **`java.util.Hashtable`** implements all the functionality of the hash table presented in the previous section. The core class also takes the next step and adds functionality that can be useful to the programmer. This section discusses this additional functionality.

The Java **`Hashtable`** class generates the hash code value by using the **`hashCode()`** method internal to each object supplied as a key. The sample hash table supplied a specific **`hashCode()`** method that was a part of the **`HashTable`** class to be used for all keys. This method enables the table to convert any object supplied as a key into an appropriate hash code. It is not necessary to restrict the universe of keys to strings, as in the example. The default

hashCode() method supplied by the **Object** class does not guarantee that the hash code values will be positive. If we are using the default hash codes to index into the array, we must make the hash code non-negative for the calculation. The Java hash table does this by performing a bitwise **AND** on the value that causes the leftmost bit in the int to be dropped if it is set. Java uses this leftmost bit to determine the sign of an int. The int values from 0x0 to Ox7FFFFFFF are the positive integers 0 to 2147483647. The int values from 0x80000000 to 0xFFFFFFFF are -2147483648 to -1.

The Java **Hashtable** class uses a data class similar to the **HashObject** in this example. It is called the **HashtableEntry** class. In addition to storing the **key**, **value**, and **next** fields, it also stores the hash code for the key and provides a **clone()** method.

Figure 4-7
The setting of the leftmost bit indicates a negative integer number

int value	hexadecimal value	binary value
2147483647	7FFFFFFF	01111111111111111111111111111111
-2147483648	80000000	10000000000000000000000000000000
-1	FFFFFFFF	11111111111111111111111111111111

In the Java core **Hashtable** class, constructors enable the user to define the size of the initial table and the percentage filled before rehashing takes place, as shown in the following code:

```
public Hashtable(int initialCapacity, float loadFactor)
public Hashtable(int initialCapacity)
public Hashtable()
```

If one or both of these values are not specified, the class defines defaults for them. A default Hashtable object has 101 buckets and is rehashed when it is 75 percent full. The Java **Hashtable** class, by default, keeps an average of less than one key per bucket. It is up to the user of the class to decide the best values for these fields for the implementation. The user cannot change the **loadFactor** and **initialCapacity** after the table is constructed.

The **size()** and **isEmpty()** methods are accessor methods used to determine the actual number of key/value pairs stored in the hash table, as shown in this sample code:

```
public int size()
public boolean isEmpty()
```

If there are no elements, **isEmpty()** returns **true;** otherwise, it returns **false.** The **size()** method simply returns the value of the **count** field.

The **keys()** and **elements()** methods supply an enumeration of the hash table data:

```
public synchronized Enumeration keys()
public synchronized Enumeration elements()
```

An *enumeration,* in this context, is a traversable list of objects. The **keys()** method supplies

a list of all keys in the table, and the **elements()** method supplies a list of all data objects stored in the table.

The **contains()** and **containsKey()** methods search the hash table for the desired object, as shown in this code:

```
public synchronized boolean contains(Object value)
public synchronized boolean containsKey(Object key)
```

The **contains()** method traverses the entire table looking for a match to the supplied argument in the **value** (data) field. The **containsKey()** method uses the hash table lookup to find the desired **key** object. Both these methods use the objects internal **equals()** method to determine whether a match has occurred.

The **get** and **put** operations are supported much like the **get()** and **put()** methods in this example:


```
public synchronized Object get(Object key)
public synchronized Object put(Object key, Object value)
```

The Java **Hashtable** class adds more error checking to these operations. The **put()** method checks that the data object is not **null.** The **get()** method returns **null** if the key is not found.

The Java hash table includes **remove()** and **clear()** methods to enable the deletion of one or all of the key/value pairs in the table. The **remove()** method takes a key for an argument. The key is searched for in the table, and, if it is found, it is removed from the table returning the value stored for that key. The method returns **null** if the specified key is not found. The **clear()** method removes all the entries in the table but does not affect the capacity:

```
public synchronized Object remove(Object key)
public synchronized void clear()
```

The **Hashtable** class provides a shallow copy **clone** method to create a duplicate hash table. The standard **toString()** method is implemented to create a string that includes a complete list of all keys and data values stored in the table. The keys and values are represented by their internal **toString()** methods, as shown in this code:

```
public synchronized Object clone()
public synchronized String toString()
```

Two very handy methods provided by the **Hashtable** class are **readObject()** and **writeObject()**. These methods enable the user to save and restore the entire contents of a hash table through a stream. The process of sending an object through a stream is called *object serialization.* The methods do not save or restore the actual state of the hash table capacity and load factor The hash table restored will not necessarily be equivalent to the table saved, except that both have the complete set of all the key/value pairs. The number of buckets and the specific buckets in which the entries reside may differ. Using **readObject()** and **writeObject()** gives the programmer a way to make the data stored in the hash table persistent from one run of the program to the next, as shown in this code:

```
private synchronized void writeObject (java.io.
     ObjectOutputStream s)
private synchronized void readObject (java.io.
     ObjectInputStream s)
```

The **`rehash()`** method, for the most part, works the same as that in the example, except that the capacity is hard coded to be approximately doubled each time the table is rehashed:


```
protected void rehash()
```

Uses of the Hash Table

As we've seen, hash tables can be used to store any type of objects, as long as some unique key exists for each value stored. The key can be an object of any class, as can the data. One possible use for a hash table is to store information associated with points in a Cartesian coordinate system. Normally, a two-dimensional array might be used in this case, but what if the data is only for a small number of points in the map?

In a 200x200-pixel map such as this, it would take a two-dimensional array of dimensions [200][200], or 40,000 data objects, to store the information for all possible points. Even though all 40,000 objects do not necessarily

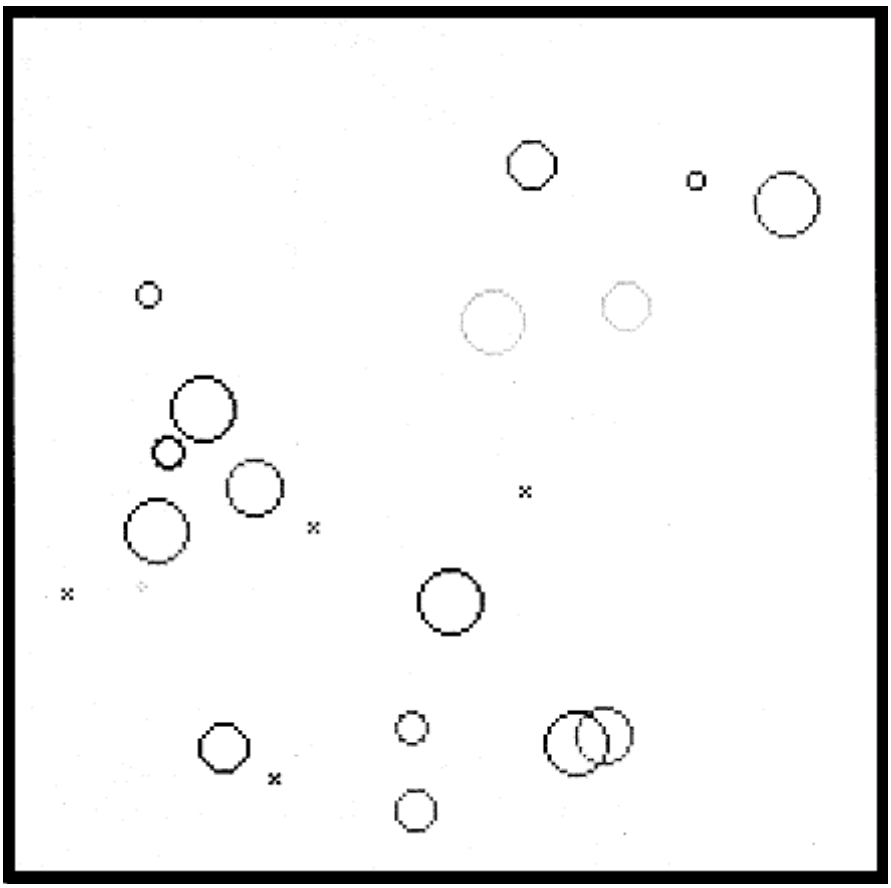


Figure 4-8
The output from the PointTest program.
The entire area represents a 200X200 point grid.

need to be allocated, the array itself must be allocated from the free memory space. The hash table can be a viable alternative in this case. It only needs to allocate memory based on the number of points saved. Figure 48 was generated using the code in Figure 4-9.

`PointInfo` is the data class for the PoinTest application. The source code for **`PointInfo`** is shown in Figure 4-10.

Figure 4-9
PointTest.java.

```
package adt.Chapter04;

import java.awt.*;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Random;

public class PointTest
    extends Frame
{
    public PointTest()
    {
        Point p;
        PointInfo pi;
        int x;
        int y;
        Color color[] = { Color.green, Color.blue,
              Color.red, Color.black };

        table = new Hashtable( 23, 2.0f );
        rand = new Random();

        for( int i = 0; i < 50; i++ )
        {
            x = Math.abs( rand.nextInt() % 200 );
            y = Math.abs( rand.nextInt() % 200 );
            p = new Point( x, y );
            pi = new PointInfo( color[ i%4 ],
                 (rand.nextInt() % 20) + 1 );
            table.put( p, pi );
        }

        resize( 200, 200 );
    }

    public void paint( Graphics g )
    {
        Point p;
        PointInfo pi;
        Enumeration e = table.keys();

        while( e.hasMoreElements() )
```

Continues

Figure 4-9
Continued.

```
        {
            p = (Point)e.nextElement();
            pi = (PointInfo)table.get( p );

            g.setColor( pi.getColor() );
            g.drawOval( p.x, p.y, pi.getSize(),
                        pi.getSize() );
        }
    }

    public static void main( String args[] )
    {
        PointTest p = new PointTest();
        p.show();
    }

    Hashtable table;
    Random rand;
}
```

Figure 4-10
PointInfo.java.

```
package adt.Chapter04;

import java.awt.Color;

public class PointInfo
{
    public PointInfo( Color color, int size )
    {
        this.color = color;
        this.size = size;
    }

    public Color getColor()
    {
        return color;
    }

    public int getSize()
    {
        return size;
    }

    public void setColor( Color color )
    {
        this.color = color;
    }
```

```
    public void setSize( int size )
    {
```

Continues

Figure 4-10
Continued.

```
        this.size = size;
    }

    Color color;
    int size;
}
```

Properties as a Subclass of the Hash Table

The Java core classes define the Properties class as an extension (subclass) of the **`Hashtable`** class. The **`Properties`** class is a hash table in which all the keys and data are in the form of String objects. This table can be used to store and access configuration and environment settings, as well as to perform other functions.

A system-level **`Properties`** object is in every Java application or applet. This object contains the key/value pairs that describe the attributes of the system under which the Java program is running. This object can be queried to determine any platform-specific information available. A simple, one-line Java program can demonstrate the properties available. The source code for the **`PropertyList`** application follows:

```
class PropertyList
{
    public static void main( String arg[} )
    {
        System.getProperties().list( System.out );
    }
}
```

Figure 4-11 shows the output generated from running this program.

The **`Properties`** class has a couple of important distinctions from its superclass, **`Hashtable`.** As mentioned earlier, the **`Properties`** table holds key/value pairs consisting entirely of **`String`** objects. These pairs can be loaded directly from a text file opened as a stream. The text file contains entries of this form:

```
KEY1=VALUEl
KEY2 VALUE2
...
KEYn=VALUEn
```


Figure 4-1 1

Ouput from the PropertyList application.

```
-- listing properties --
java.home=C:\Java\JDK
awt.toolkit=sun.awt.win32.MToolkit
java.version=internal_build
file.separator=\
line.separator=


java.vendor=Sun Microsystems INC.
user.name=mjenkins
os.arch=x86
os.name=windows 95
java.vendor.url=http://www.sun.com/
user.dir=C:\Java\Developement\src
java.class.path=.;C:\Java\JDK\lib\Classes.xip;C:\Java...
java.class.version=45.3
os.version=4.0
path.separator=;
user.home=C:\Java\JDK
```

A text file like this is extremely useful for reading or writing configuration files. The **`Properties`** class also can supply default values for keys that are not set explicitly in the table. The **`Properties`** constructor creates an empty hash table. It also can reference a defaults hash table, as shown in this code:

```
public Properties()
public Properties(Properties defaults)
```

The **`load()`** and **`save()`** methods provide the mechanism for reading or writing a text configuration file (stream), as shown in this code:

```
public synchronized void load(InputStream in)
public synchronized void save(OutputStream out,
      String header)
```

The **`getProperty()`** method returns the string value associated with the string **`key`** parameter. If the **`key`** is not found in the **`Properties`** table, it is supplied by the default Properties table (supplied to the constructor). If the **`key`** is not found in the defaults either, it uses the **`defaultValue`** parameter to the **`getProperty()`** method if one is supplied:

```
public String getProperty(String key)
public String getProperty(String key, String defaultValue)
```


The **`propertyNames()`** method returns an enumeration of all keys in the hash table. The **`list()`** method takes a **`PrintStream`** argument to which it dumps the entire **`Properties`** list:

```
public Enumeration propertyNames()
public void list(PrintStream out)
```

This is handy for debugging purposes, although the maximum **`String`** size it prints is 40 characters. After that, the value **`String`** is truncated, and an ellipsis is appended to the

`String`.

Using Properties To Pass Command-Line Information

The Java runtime process (**`java.exe`** on Windows 95, **`java`** on UNIX) also is tied to the **`Properties`** class. One of the available command-line arguments to the runtime process is the **`-D`** option. The **`-D`** option defines a key/value pair in the system's **`Properties`** object, as shown in this example:

```
java -Dthis.is.my.property=HELLO HashTest
```

The runtime will put an entry into the system's **`Properties`** object with the key "**`this.is.my.property`**" and the value "**`HELLO`**". A call to the following, for example, returns "**`HELLO`**":

```
System.getProperties() .getProperty("this.is.my.property")
```

We can specify as many **`-D`** properties on the command line as we want. If the number gets too large, though, it usually is better to create a configuration file and load a **`Properties`** object from the file.

Exercises

1. Rewrite the **`HashTable`** sample class to include the following:

A **`remove()`** method to delete a specific key/value pair.

A **`clear()`** method to remove all keys from the table.

A constructor that lets the user set the initial capacity, rehash size, and capacity increment.

A built-in **`hashCode()`** method inherited from the key's class.

A **`stats()`** method that supplies the number of entries in the hash table as well as the number of keys in each bucket.

2. Modify the **`NoSuchKeyException`**'s message string to include the hash code for the missing key and a list of all keys in the bucket to which the missing key should belong.

3. Write a Java application that generates a report showing the *complete* set of system properties (do not use **`Properties.list()`**, because it only displays the first 40 characters of the value).

4. Write a Java application that reads in a configuration file similar to the following and also displays the properties set:

```
NAME=my name
DEFAULTDIR=C:\TMP
VERSION= 1.0
EMAIL=my email address
WWW=my home page URL
```

Summary

In this chapter, we learned the following:

- That a hash table is similar in many ways to a standard array
- How to create basic hashing algorithms
- How to look up values in a hash table based on the key by which it is stored
- The process by which a hash table is resized by moving all its entries into a new, larger table
- How to use the default Java **`hashCode()`** method to generate hash codes
- How to use the **`Properties`** class to examine the available system information at runtime
- How to use the **`Properties`** class to use a configuration file

Chapter 5
Linked Lists

In this chapter, we'll examine the linked list. *Linked lists* are container types that store collections of data in a sequential order. The concept of a generic data node also is introduced and explained. The standard linked list operations are covered in detail, and examples are given for simple add, insert, and delete methods. Both array-based and non-array linked list implementations are examined and contrasted. List traversal is explained and implemented using the **`java.util.Enumeration`** interface.

To demonstrate the use of a linked list, we'll develop a simple phone book utility. This utility will use a linked list to store name information. Each of the linked list implementations developed in this chapter will be used in the phone book utility to examine the linked list functionality.

The Linked List as a Base ADT

When *abstract data types* (ADTs) are mentioned to programmers, the first thing that probably comes to their minds is the linked list. A simple linked list is the basis for many more advanced ADTs—some of which are examined later in this book. Why is the linked list as important as an ADT? It encapsulates one of the most common programming tasks: the organization and maintenance of a dynamic collection of data in a specified order. The linked list, in its many

incarnations, is well suited to this task.

In Chapter 4, "Hash Tables," we looked at the operations of a hash table and learned that, at a minimum, this table includes **`put`** and **`get`**. In this chapter, we'll define a minimal set of operations on the linked list to be **`add`**, **`insert`**, **`get`**, and **`delete`**. Linked lists are empty when created. While using a linked list, elements can be added to the list end of the list and inserted at specific locations in the list, or later deleted from the list. And, of course, at some point, the user will need to retrieve the objects stored in the list.

In some ways, a linked list is similar to a vector. Elements are stored in a specific order and can be accessed in that same order. For the most part, though, all operations on a vector container are based on the index of the item in the vector array. In a linked list, the operations are based strictly on sequential access to the elements in the linked list instead of on an index. Access to an element in the linked list is based on linear traversal of the list, similar to the way in which the enumeration that the vector's **`elements()`** method provides its **`hasMoreElements()`** and **`nextElement()`** methods. Also, Java vectors are based on storage in a contiguous array. In a linked list, the data stored also can be stored in contiguous memory, although this is not a requirement. This brings up one of the possible advantages of using a linked list: Growing a vector requires the creation of a new, larger internal array and then the copying of the old data into the new array. It is possible to implement a linked list in which the memory for each element is created at the time the element is stored. This ensures that no extra memory space is wasted.

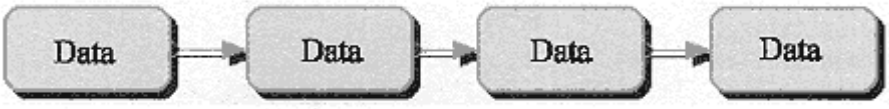


Figure 5-1
Four data nodes in a linked list.

A linked list can be represented as a collection of data objects tied together in a chain, with each object keeping track of the next object in the list. In Figure 5-1, each data object in the chain knows how to access the next data object. As long as we know how to access the first object in the list, we can access any of the objects in the list simply by walking the chain! The job of the linked list is to create and maintain this data chain and to give users a way to walk the chain.

Now we'll take a minute to define our requirements so far for the linked list. We need the capability to store a collection of generic data items in order. We must be able to traverse the list and access each element in that same order. We need to be able to add data items to the list by appending, prepending, or inserting them so that the order we choose can be maintained. We also need to be able to delete a particular item from the list.

An Array-Based Linked List

To implement our linked list, we need to create an ordered collection of elements. One way to achieve this is to use an array as the collection container. Indexing the array provides us with our mechanism to find the next element in the chain. We simply add 1 to the index to find the

"next" element. In Java, we can create an array of primitive or reference types with the new ([ ]) operators. As we saw earlier, though, the Java **`vector`** class is a safe, generic way to create a dynamically sized array. Using the vector gives us the immediate advantage of freeing us from the need to know the size of the list ahead of time. We also have many of the operations we will need predefined, such as the capability to insert an object into the vector at a specific index or to delete a specific item from the vector. As is often the case, reusing and extending the Java core classes is a good way to reduce development time and increase productivity.

Now take a look at the simple implementation of a vector-based, singly-linked list class in Figure 5-2. Here, we provide one public constructor that creates an empty list and initializes our instance variables. The *instance variables* are references to the vector we use to store the data (**`dataSet`**) and

Figure 5-2
VSLinkedList.java.

```
package adt.Chapter05;


import java,util.Vector;
import java.util.Enumeration;

public class VSLinkedList
{
     public VSLinkedList ()
     {
          current = 0;
          dataSet = new Vector ();
     }

     public void add( Object o )
     {
          dataSet.addElement(o);
          current = dataSet.size() - 1;
     }

     public void insert( Object o )
     {
          dataSet.insertElementAt( o, current );
     }

     public void delete()
     {
          dataSet,removeElementAt(current);
          if( current >= dataSet.size () )
               current--;
     }

     public void setCurrent( Object o )
     {
          dataSet.setElementAt(o,current);
     }
```

```
    public Object getCurrent()
    {
        return dataSet.elementAt(current);
    }

    public void reset()
    {
        current = 0;
    }

    public boolean next()
    {
        if( ++current < dataSet.size() )
        {
            return true;
```

Continues

Figure 5-2
Continued.

```
        }
        else
            return false;
    }

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

    protected Vector dataSet;
    protected int current;
}
```

an int primitive, which we use to keep track of our current position in the linked list, **`(current).`** In Figure 5-2, we declare our instance variables in this case to be protected to allow the extension of our **`VSLinkedList`** class. Any subclasses we create that use **`VSLinkedList`** as a superclass most likely will need access to these variables.

Most of the methods in the **`VSLinkedList`** class consist of one statement. This is because we are taking advantage of the vector's functionality to do most of the work for us. Now take a look at what each method does.

The **`add()`** method adds an element to the end of the list by using the **`Vector.addElement()`** method. Additionally, the **`current`** field needs to be adjusted. The field always should point to the last element that was affected. In this case, that corresponds to the last element in the list.

The **`insert()`** method inserts the new element before the element indicated by the **`current`** field. The vector **`insertElementAt()`** method is used. The value of the **`current`** field doesn't need to change, because it now indicates the position of the new element.

The **delete()** method uses the **Vector.removeElementAt()** method to delete the element at the **current** position in the list. Once again, the **current** field doesn't need adjustment, because it now indicates the position of the element in the list that occupies the position vacated by the deleted element. If no other references exist to the deleted element, its memory is freed on the next pass of the garbage collector.

The **setCurrent()** method is used to update or replace the data in the **current** position in the list. The element is replaced by using the **Vector.setElementAt()** method. The reference to the displaced element is lost to the list and, like the deleted element, its memory is freed on the next pass of the garbage collector if no outside references exist to the element.

The **getCurrent()** method is used to retrieve the element stored at the current position in the list. It uses the **Vector.elementAt()** method to return the object. It is the responsibility of the caller of the method to cast the returned object to the appropriate type.

The **reset()** method and the **next()** method usually work together. They enable the user to manipulate the **current** field of the linked list. The **reset()** method forces **current** to the first position in the list. The **next()** method then is used to move the **current** position forward in increments of 1. Users therefore can walk through the list and perform any of the other operations they want. **next()** also returns a boolean value to indicate whether it has reached the end of the list. If **next()** returns **false**, **current** is set to the position of the last element in the list.

Putting the Linked List to Work

Now that we've developed a linked list class, it's time to put it to work. In this section, we are going to write an elementary Java application we can use to put the linked list through its paces. We will develop an address book application that keeps track of names and prints a listing.

Because the address book will be nothing more than a specific implementation of our linked list, we will derive the **AddressBook** class directly from **VSLinkedList**.

Remember that a subclass inherits all the public and protected methods of its superclass. So, for the most part, all we need to do is write the **main()** method. This is an application rather than an applet. We will not need to write any HTML code, and we won't need to use a browser to run the application. For this example, we are not interested in presenting a nice GUI interface, so we can just use the console for output.

To run the test, we can have our address book perform the following steps:

1. Create a linked list.

2. Add a list of names to the list.

3. Print the entire list.

4. Verify that the list is printed in the same order as entered.

5. Insert a new name at an arbitrary location in the list.

6. Reprint and verify the insertion.

7. Delete a different name from the list.

8. Reprint and verify the deletion.

The **`AddressBook`** class **`main()`** method creates the new **`AddressBook`** object and runs right through the steps as outlined. Five names are added to the phone book (without phone numbers), and the list is printed. To print the list, though, we need to walk through the entire chain and print each name as we go. Luckily, the linked list class has just the thing we need; the **`reset()`** and **`next()`** methods are made for this type of work. We use these methods as the basis for our **`print()`** method.

Figure 5-3
AddressBook.java.

```
package adt.Chapter05;

import java.util.Enumeration;

public class AddressBook
    extends VSLinkedList
{
    public static void main( String args[] )
        throws java.io.IOException
    {
        AddressBook addrBook = new AddressBook();
        addrBook.add( "Jim Jones" );
        addrBook.add( "Mike Smith" );
        addrBook.add( "Patty Thompson" );
        addrBook.add( "Joan Barker" );
        addrBook.add( "Joe Block" );

        addrBook.print();

        addrBook.reset();
        while( !( ((String)addrBook.getCurrent()).equals
                ( "Joan Barker" ) ) )
        {
            addrBook.next();
        }
        addrBook.insert("John Smith");
        addrBook.print();

        addrBook.reset();
        while( !(((String)addrBook.getCurrent()).equals(
                "Mike Smith" )) )
        {
            addrBook.next();
        }
        addrBook.delete();
```

Continues

Figure 5-3
Continued.

```
        addrBook.print();
        System.in.read();
    }

    public void print()
    {
        reset();
        do
        {
               System.out.println( getCurrent() );
        } while( next() );
    }
}
```

Figure 5-4
The output for our address book application.

```
Jim Jones
Mike Smith
Patty Thompson
Joan Barker
Joe Block
Jim Jones
Mike Smith
Patty Thompson
John Smith
Joan Barker
Joe Block
Jim Jones
Patty Thompson
John Smith
Joan Barker
Joe Block
```

Then we walk through the list to find an entry that matches "Joan Barker". After "Joan Barker" is set to **`current`**, insert a new node for "John Smith". Reprint the list with the **`print()`** method.

Finally, we reset the list again and walk through it to find "Mike Smith". When we find his node, we delete it and again call **`print()`**. Here is the complete class definition for the **`AddressBook`** application:

Figure 5-4 shows the output for this program.

This linked list was very easy to implement, because we took advantage of the vector's built-in functionality, reducing our development time. In doing so, however, we also inherit the

overhead of the vector. The vector-based implementation of the linked list has several disadvantages. First, consider the impact of the implementation on system resources. The two resources impacted by our linked list are memory and CPU use. By using a Java **`vector`** object as our underlying mechanism, we inherit not

only its functionality but also its overhead. In Chapter 3, "Arrays, Vectors, and Sorting," we examined the way in which the **`Vector`** class allocates memory for the storage of its elements. If we do not actively want to manage the memory of the vector, we will have to live with the default behavior This means that every time the vector runs out of preallocated memory, its size doubles. Depending on our list, this could lead to a lot of allocated but unused memory.

We could have the list internally manage the size of the vector, but that would have the side effect of causing the vector to perform a lot of **`arraycopy()`** operations (see "Vectors versus Arrays" in Chapter 3). This process could be very time-intensive and could bring about less than optimal performance in our linked list. The insert operation also will always cause an **`arraycopy()`** to occur. Again, we could program around this behavior by always adding to the vector sequentially and keeping an index for the next node in the node itself. If we do that, however, we give up the capability to use much of the built-in functionality of the vector.

To summarize, it appears that using an array-based implementation of our generic linked list might not be the most efficient approach. Another alternative is to use a reference-based implementation to address these issues and develop a better, more efficient, and more easily extensible linked list. In the next section, we'll take a look at such an approach.

Nodes

Before we jump ahead to the new linked list, take a quick look back to the **`HashObject`** from Chapter 4. There, we used the **`HashObject`** to make a list in each bucket in the hash table. This is really a linked list in its simplest form. All the **`HashObject`** did was chain together data elements into a linked list construct. The **`HashObject`** is a construct called a *node*. We can consider a node to be one part of a larger conglomerate whole. Individual computers in a network sometimes are referred to as *nodes*.

The basic data storage container in the linked list is the node. A linked list node contains, at a minimum, a placeholder for our stored data and some mechanism to reference the next node in the chain. A node container also provides a layer of abstraction between the user and the implementation. This gives us more flexibility in designing our implementation by disassociating the data from the container. In other words, we can redesign the node in any way we want in the future without affecting the code that uses the linked list.

A node also is similar in concept to the Java core class vector element we examined in Chapter 3. Unlike a standard array, the user has no direct access to the individual nodes. The user interacts with the data through accessor methods defined in our node API. Figure 5-5 shows a simple node base class.

Figure 5-5

SLNode.java.

```
package adt.Chapter05;

class SLNode
      implements Cloneable
{
    protected SLNode()
      {
            data = null;
      }

    protected SLNode( Object data )
      {
            setData(data);
      }

    protected void setData( Object data )
      {
            this.data = data;
      }

    protected Object getData()
      {
            return data;
      }

    protected void setNext( SLNode next )
      {
            this.next = next;
      }

    protected Object getNext()
      {
            return next;
      }

    public boolean equals( Object o )
      {
            return data.equals(o);
      }

      private Object data;
      private SLNode next;
}
```


Figure 5-6
shallowCopy versus deepCopy.

```
class Foo
{
     public static Foo shallowCopy( Foo a )
     {
          Foo b = new Foo();
```

```
            b.s1 = a.s1;                    // Copy the
                 reference
            return b;
       }

       public static Foo deepCopy( Foo a )
       {
            Foo b = new Foo();

            b.s1 = new String( a.s1 ); // Copy the object
                 (String)
            return b;
       }

       String s1;
}
```

The first thing we might notice about the **SLNode** (*Single Link Node*) class is that we did not declare it to be public. According to the Java visibility rules, this class is accessible only to classes in the same package. We limit the visibility of our **Node** class, because the mechanism we use for data storage and manipulation is implementation specific. These types of details should be hidden from the end user of our linked list. This kind of encapsulation gives the programmer the freedom to change the implementation later without affecting any existing code that uses the linked list.

We have declared that the **SLNode** class implements the **Cloneable** interface. The **Cloneable** interface in Java defines no methods; it simply allows the **SLNode** object to be copied using the standard **clone()** method inherited from the **Object** class. Any class can declare that it implements **Cloneable** to enable the use of the **clone()** method. The default **Object.clone()** is a native (operating-system dependent) method that performs a shallow copy of the instance object. A *shallow copy* means that each member is copied into the corresponding member of the new class instance. In other words, the reference to a member is copied, but a new copy of the member object is not created (see Figure 5-6). We can override the **clone()** method to specialize the cloning behavior if it becomes necessary to have a deep copy. A *deep copy* is a copy in which the members of the class are copied as well.

The class provides two constructors: a default constructor to create an empty node and a constructor to create an **SLNode** and initialize the data

field. Neither of the constructors needs to be declared as public, because the class cannot be instantiated outside of the package.

SLNode uses the **Object** class as the internal data type. As the base class for all Java classes, this provides the generic behavior we want in an ADT. Any Java reference or array type can be stored in our **SLNode.** Primitive types can be stored in our **SLNode** by using the standard Java wrapper classes. We declare the data field member as **private** to provide data encapsulation to the class. All access to the data field must come through the accessor methods **getData()** and **setData().**

Also, an instance variable **next** is used to store a reference to the next node in the chain. In

our vector-based implementation, we were using the vector's built-in indexing to keep track of the position. **SLNode** also provides two accessor methods to allow outside access to this field. So now we have a means to reference the next node in the chain without having to rely on the vector's behavior.

A Reference-Based Linked List

Now that we have created the data storage mechanism for our linked list, we need to chain together these nodes to create our linked list. This type of linked list is called a *reference-based linked list.* The name stems from the fact that the node objects use a reference to track the next object in the list. To complete the linked list, we need to provide the add, insert, and delete functionality.

Now take a look at the **SLinkedList** class; it provides a single constructor to create an empty **SLinkedList** and initialize the instance variables. The only instance variables defined in this case are **head** and **current,** which are both SLNodes. **head** keeps track of the first node in the chain, and **current** is used as an internal placeholder. Now it's time to take a look at the meat of the linked list. Take a look at each of the operations and their implementations for the linked list in Figure 5-7.

Standard Linked List Operations Revisited

The **add()** method is used to append a data element to the list. It first needs to create a new node object to hold the data. Next, we need to determine whether the list is empty. If **head** is **null,** we haven't yet added

Figure 5-7
SLinkedList.java.

```
package adt.Chapter05;

import java.util.Enumeration;

public class SLinkedList
{
     public SLinkedList()
     {
          head = null;
     }

     public void add( Object o )
     {
          SLNode newNode = new SLNode(o);

          if( head == null )
          {
               current = head = newNode;
               return;
          }

          current = head;
```

```
        while( current.getNext() != null )
             current = current.getNext();
        current.setNext( newNode );
        current = newNode;
             }

             public void insert( Object o )
             {
                   SLNode newNode = new SLNode(o);
                   SLNode tmp = current.getNext();
                   current.setNext( newNode );
                   newNode.setNext( tmp );
             }

             public void delete()
             {
                   SLNode tmp = head;

                   if( current == head )
                   {
                        current = head = head.getNext();
                        return;
                   }

                   while( tmp.getNext() != null )
                   {
                         if( tmp.getNext() == current )
                   {
                         tmp.setNext( current.getNext() );
```

Continues

Figure 5-7
Continued.

```
                         current = tmp.getNext();
                         return;
                   }
             }
        }

public void setCurrent( Object o )
{
      current.setData(o);
}

public Object getCurrent()
{
      return current.getData();
}

public void reset()
{
      current = head;
}
```

```
    public boolean next()
    {
        current = current.getNext();
        if( current == null )
            return false;
        else
            return true;
    }

    public SLEnumeration elements()
    {
        return new SLEnumeration( head );
    }

    protected SLNode head;
    protected SLNode current;
}
```

any nodes; all we need to do is set **head** and **current** to reference the new node we created. We always want to set **current** to reference the last node we affect. This enables us to keep track internally of where we are. We will find this especially useful in the **insert()** and **delete()** methods. If the list is not empty, we need to find the end of the list and add the node there. We accomplish this by setting **current** to the start of the list **(head)** and walking the list until **next** is **null.** When we find the end of the list, we simply let **next** refer to the new node.

The **insert()** method is a little different. It is used to insert a data element into the list at the current position. We still need to check to see whether the list is empty. If it is, the users really aren't inserting—they are adding. This is not an important enough distinction to the users of the linked list, so there is no need to throw an exception. We'll just handle the situation internally by calling the **add()** method. We could just duplicate the **add()** code in the insert, but if we change the implementation of **add()** in the future, we will have to propagate that change to **insert()** as well. If the list is not empty, we need to break the existing chain at the current node and rejoin it with our new node in place, as shown in Figure 5-8. We assume that the user has positioned **current** in the proper location in the list and wants to insert immediately before the node that **current** references.

The **delete()** method also is concerned with checking for an empty list. The only case where **current** should ever equal **null** is when the list is empty. We just as easily could check for **head == null.** Because the operation is based on the position of the **current** node, though, we'll check **current** to be consistent. It also is possible that the user is deleting the first node in the chain. This case requires special handling as well, because we have to reset the **head** reference. After we move the **head** reference to **head**'s next, the list loses all references to the old **head** object, and the garbage collector should free its memory on the next pass.

Removing the **head** node is a special case, though; now look at a normal deletion. Normal deletion presents us with a special problem of its own. To delete a node, we need to break the

chain around the node we want to delete and restore the chain excluding the deleted node. To do this, we need to know what the previous node was to restore the link! But our node only keeps track of the next node in the series. To get around this problem, we need to walk the list until we find the node where **`next`** refers to **`current`.** Then we set the previous node's **`next`** to **`current`**'s **`next`**, and **`current`** drops out of the chain. All that remains is to reset **`current`** to refer to the old **`current`**'s **`next`,** and we're done.

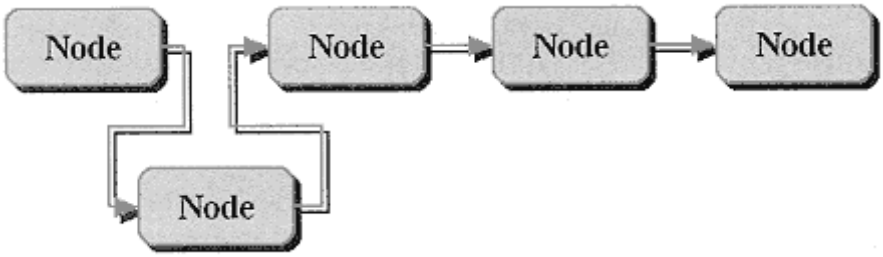


Figure 5-8
A new node is inserted into the list by rearranging
the references of the previous node.

Because we have an internal placeholder (**`current`**), and we base our **`insert()`** and **`delete()`** operations on the "current" node, we need to provide a mechanism for the user to walk through the list to find the desired node. To provide this functionality, we've supplied three public methods: **`reset()`**, **`next()`**, and **`getCurrent()`**. The user can use these three methods in combination to identify any node in the list. This is exactly like our vector-based implementation.

The procedure starts by resetting the list. Then, it repeatedly tests the current node and steps to the next node if necessary. All three methods are extremely simple. The **`reset()`** method just sets **`current`** to **`head`** so that we can begin at the start of the list. The **`getCurrent()`** method returns a reference to the data stored in the current node, and **`next()`** moves **`current`** by one node after first checking to see that we are not going to step off the end of the list. This extra check in the **`next()`** method also ensures that **`current`** will never be **`null`** unless the list is empty.

We've also supplied a **`setCurrent()`** method to enable the user to update the data at any node using the same type of procedure, and we have an **`isEmpty()`** method to enable the user to determine whether the list is, in fact, empty. We determine that the list is empty if **`head`** refers to **`null`**.

List Traversal

One of the advantages of storing data in a linked list is that it is relatively easy to perform a repetitive operation on the data stored in this fashion. This kind of processing is commonly known as *list traversal* or *enumeration.* In Java terminology, it is called *enumeration,* and an **`Enumeration`** interface is defined for implementation by any class for which enumeration is appropriate.

Remember that a Java interface defines only the names and the signatures of the methods

required. It is up to the implementing class to actually define the method. We can use two basic approaches to implement the **`Enumeration`** interface for a linked list. We can declare that our linked list implemented the interface internally, or we can define a separate class in which the sole purpose is to enumerate a linked list object.

If we opt to implement the interface within the linked list, we will have to create another placeholder that is similar to **`current`**, or we'll need to use **`current`** as our enumeration placeholder. It's not a good idea to reuse **`current`** in this capacity, because this might interfere with other opera-

tions on the list and could confuse the linked list user. After we define this additional placeholder, we just need to implement the two methods defined by the interface: **`hasMoreElements()`** and **`nextElement()`**.

For this example, we'll use the second method. It's a better object-oriented solution, because it enables us to disassociate the enumeration from the linked list implementation. It also gives us the capability to create multiple instances of the enumeration and simplifies the further extension of the **`Enumeration`** class.

All we need to do to create the **`SLEnumeration`** class is provide the capability to walk the data chain and determine when we are finished.

Figure 5-9
SLEnumeration.java.

```
package adt.Chapter05;

import java.util.Enumeration;

public class SLEnumeration
     implements Enumeration
{
     public SLEnumeration( SLNode first )
     {
           if( first == null )
               throw new NullPointerException("List is
empty");
           current = first;
     }

     public boolean hasMoreElements()
     {
           return (current != null );
     }

     public Object nextElement()
     {
           Object o = current.getData();
           current = current.getNext();
           return o;
     }
```

```
    public void print()
    {
        while( hasMoreElements() )
        {
            System.out.println( nextElement() );
        }
    }

    SLNode current;
}
```


The question is how to access the linked list to provide this functionality in a separate object. Well, we know that a linked list is really just a collection of nodes in which each node knows how to access the next. Again, as long as we have access to the first node in the list, we have access to all nodes in the list, but in one direction only—forward. This satisfies the requirements of the **`Enumeration`** interface, which is defined to walk through the enumeration once and only once. So, all our **`SLEnumeration`** class needs to have is access to the first node in the list: **`head`**. The class constructor therefore is defined to take an SLNode as an argument.

It might be reasonably expected that the enumeration is used to invoke some common function on each node in the list in turn, such as printing. The user could do that by implementing something similar to Figure 5-10.

In fact, because this is likely to be a requested feature, we defined the method in the Enumeration class.

Using the Reference-Based Linked List

Now we can run an application such as **`AddressBook`** with our new reference-based linked list. But first, we'll spruce up the **`AddressBook`** application a little. The first thing we should do is add address entries to the data. After all, it *is* an address book. A simple **`string`** object no longer is sufficient or practical to use when storing compound data. We also can make the address book interactive. We need to define an **`AddressEntry`** class to use as a data object.

The **`AddressEntry`** class has two constructors. One creates an empty **`AddressEntry`**, and the other fills in all the fields. The data fields are for the first name, last name, address, city, state, and ZIP code. Accessor methods

Figure 5-10
Using Enumeration to print a linked list

```
SLinkedList list = new SLinkedList();
. . .
Enumeration e = list.elements();
while( e.hasMoreElements() )
{
    System.out.println( e.nextElement() );
}
```

Figure 5-11
AddressEntry.java.

```
package adt.Chapter05;


public class AddressEntry
{
	public AddressEntry()
	{
		this.first = null;
		this.last = null;
		this.address = null;
		this.city = null;
		this.state = null;
		this.zip = null;
	}

	public AddressEntry( String first, String last, String
		address, String city, String state, String zip )
	{
		this.first = first;
		this.last = last;
		this.address = address;
		this.city = city;
		this.state = state;
		this.zip = zip;
	}

	public String getFirst()
	{
		return first;
	}

	public String getLast()
	{
		return last;
	}

	public String getAddress()
	{
		return address;
	}

	public String getCity()
	{
		return city;
	}

	public String getState()
	{
		return state;
	}
```

Continues

Figure 5-11
Continued

```
public String getZip()
{
      return zip;
}
public void setFirst( String first )
{
      this.first = first;
}

public void setLast( String last )
{
      this.last = last;
}

public void setAddress( String address )
{
      this.address = address;
}

public void setCity( String city )
{
      this.city = city;
}

public void setState( String state )
{
      this.state = state;
}

public void setZip( String zip )
{
      this.zip = zip;
}

public boolean lastEquals( String last )
{
      return this.last.equals( last );
}

public String toString()
{
    return first + " " + last + "\n" +
         address + "\n" +
         city + ", " + state + " " + zip;
}

private String first;
private String last;
private String address;
private String city;
private String state;
```

```
      private String zip;
}
```


encapsulate and give us safe access to the data fields. There is also a method to enable the user to compare for a match on the last name field. A more complete implementation would have comparison methods for all the fields.

Now we modify the **`AddressBook`** class to use the new data object and display an interactive menu. One important thing to mention is that there is an additional field in the **`AddressBook2`** class. The **`DataInputStream.in`** is used to allow the class to process the **`System.in`** stream as buffered line input. We used **`System.out`** previously to print output to the console. **`System.in`** is the standard console input stream (a **`java.io.InputStream`** object) supplied to each application. An **`InputStream`** allows for only a fairly primitive input, though, so we've created a **`DataInputStream`** that will do some of the work in processing the stream. The net effect is that the **`AddressBook2`** class can use the **`readLine()`** method to process input. Here is the complete class definition for the **`AddressBook2`** application:

Figure 5-12
AddressBook2 Java.

```
package adt.Chapter05;

import java.util.Enumeration;
import java.io.*;
public class AddressBook2
{
     public AddressBook2()
     {
          list = new SLinkedList();
          in = new DataInputStream( System.in );
     }

     public void menu()
     {
          boolean isDone = false;
          String choice = "";

          while( !isDone )
          {
                System.out.println("A Add Entry");
                System.out.println("F Find Entry");
                System.out.println("I Insert Entry");
                System.out.println("D Delete Entry");
                System.out.println("P Print List");
                System.out.println("Q Quit");
                System.out.print( "Choice: " );
                System.out.flush();
                try
                {
                      choice = in.readLine();
```

Continues

Figure 5-12
Continued.

```
                }
                catch( IOException e )
                {
                    System.out.println( "Input Exception:
                    " + e );
                }

                switch(choice.charAt(0))
                {
                     case 'a':
                     case 'A':
                                        menuAdd();
                                        break;
                     case 'f':
                     case 'F':
                                        menuFind();
                                        break;
                     case 'i':
                     case 'I':
                                        menuInsert();
                                        break;
                     case 'd':
                     case 'D':
                                        menuDelete();
                                        break;
                     case 'p':
                     case 'P':
                                        menuPrint();
                                        break;
                     case 'q':
                     case 'Q':
                                        System.exit(0);
                }
          }
     }

     private AddressEntry getEntry()
     {
          AddressEntry entry = new AddressEntry();

          System.out.print("First name: ");
          System.out.flush();
          try
          {
                entry.setFirst( in.readLine() );
          }
          catch( IOException e )
          {
                System.out.println( "Input Exception: " +
                e );
          }
```

Continues

Figure 5-12
Continued.

```
System.out.print("Last name: ");
System.out.flush();
try
{
      entry.setLast( in.readLine() );
}
catch( IOException e )
{
      System.out.println( "Input Exception: " +
      e );
}

System.out.print("Address: ");
System.out.flush();
try
{
      entry.setAddress( in.readLine() );
}
catch( IOException e )
{
      System.out.println( "Input Exception: " +
      e );
}

System.out.print("City: ");
System.out.flush();
try
{
      entry.setCity( in.readLine() );
}
catch( IOException e )
{
      System.out.println( "Input Exception: " +
      e );
}

      System.out.print("State: ");
     System.out.flush();
try
{
      entry.setState( in.readLine() );
}
catch( IOException e )
{
      System.out.println( "Input Exception: " +
      e );
}

System.out.print("Zip: ");
```

```
          System.out.flush();
```

Continues

Figure 5-12
Continued.

```
          try
          {
               entry.setZip( in.readLine() );
          }
          catch( IOException e )
          {
               System.out.println( "Input Exception: " +
               e );
          }

          return entry;
     }

     public void menuAdd()
     {
          list.add( getEntry() );
     }

      public void menuFind( )
      {
           String name;

           System.out.print("Last name: ");
           System.out.flush();
           try
           {
                name = in.readLine();
           }
           catch( IOException e )
           {
                System.out.println( "Input Exception: " +
                e );
                return;
           }

           list.reset();
           while(
              ((AddressEntry)list.getCurrent()).lastEquals(
              name )
                   != true )
           {
              if( list.next() == false )
              {
                  System.out.println("Name " + name + " not
                       found.");
                  return;
              }
           }
```

```
            System.out.println( (AddressEntry)list.
            getCurrent() );
```

Continues

Figure 5-12
Continued

```
     }

     public void menuInsert()
     {
          list.insert( getEntry() );
     }

     public void menuDelete()
     {
          list.delete();
     }

     public void menuPrint()
     {
          list.elements().print();
     }

     public static void main( String args[] )
     {
          AddressBook2 book = new AddressBook2();
          book.menu();
     }

     DataInputStream in;
     SLinkedList list;
}
```

The added functionality in this version of the **`AddressBook`** makes it quite a bit larger than the original. Because this version is interactive, we must generate input prompts for the user and process that input accordingly. For **`AddressBook2`**, we've added a **`menu()`** method to give the user the available choices. After the user inputs a choice, we process it appropriately. There is a separate method for each of the menu choices except for "Quit", which causes the application to terminate. Also, a **`getEntry()`** method is used to prompt the user for the details required in the **`AddressEntry`** when we need to create a new entry in the list.

One of the advantages of object-oriented programming and design is the capability to easily replace the underlying implementation of an object. In the case of the **`AddressBook`** applications, it would have been a trivial matter to substitute the vector-based linked list for the reference-based linked list, or vice versa. This kind of flexibility is especially characteristic of the Java language. Java's interfaces encourage the definition of standard APIs. The implementations of these APIs easily can be substituted for one another, which leads to a more flexible, extensible system.

Exercises

1. Create a linked list implementation from scratch using Java arrays. Do not use the **`vector`** class.

2. Create a simple linked list class that is extended from **`SLNode`** instead of containing a list of SLNodes.

3. Extend the implementation of **`AddressBook2`** to do the following:

Supply mechanisms to look up by first name, last name, address, city, state, or ZIP code.

Allow filtered printing (only list cities that match "Chicago", for example).

Automatically insert the entries in alphabetical order by last name.

Summary

In this chapter, we learned the following:

- The linked list is a very commonly used ADT.
- Linked lists share some common attributes with vectors, arrays, and hash tables.
- We can implement a linked list construct by using the **`Vector`** class as a base.
- We learned what a node is and how to use it with the linked list.
- We learned about the **`java.lang.Cloneable`** interface.
- We learned the difference between a shallow and a deep copy.
- We implemented a reference-based linked list construct using node objects.
- We looked at using list traversal and enumeration in linked lists.
- We examined extending the standard **`java.util.Enumeration`** class to make it more useful.

Chapter 6
Circular and Doubly-Linked Lists

This chapter covers a few of the extensions to the linked list class. Better superclasses are defined, and examples explain the implementation of doubly-linked and circular-linked lists.

The impact of performance and flexibility is explored in more complex implementations. The integration of the quicksort algorithm developed in Chapter 3, "Arrays, Vectors, and Sorting" is one of the exercises presented near the end of this chapter.

Extensible Linked List Superclasses

In the last chapter, we examined the singly-linked list and looked at classes for nodes, enumerations, and the linked list itself. Unfortunately, the design of these classes isn't as flexible as it could be. One reason for this is that some implementation-specific methods and fields are included in the classes to simplify and clarify the explanations.

In the introduction to Chapter 5, we learned that we can use the linked list as the basis for several other abstract data types. The ADTs covered in the next few chapters are all derivatives of the linked list class. It therefore is appropriate to define some truly generic linked list superclasses from which to extend.

We will start out with a new **Node** superclass (see Figure 6-1). One of the things that is going to differ in the linked list derivatives is the num-

Figure 6-1
Node.java.A

```
package adt.Chapter06;

class Node
     implements Cloneable
{
     Node()
     {
          data = null;
     }
     Node( Object data )
     {
          setData(data);
     }

     void setData( Object data )
     {
          this.data = data;
     }

     Object getData()
     {
          return data;
     }

     public boolean equals( Object o )
     {
          return data.equals(o);
     }

     protected Object data;
}
```

ber and functionality of the links. The **SLNode** contains a reference to the next node in the list, appropriately called **next.** Some of the linked list types we will look at will not encompass the concept of "next", per se. To allow for flexibility in defining the links, our **Node** superclass will not specify any link at all. It will be left to the subclasses to define and implement the appropriate link mechanism for the class.

The new **Node** class is very similar to the **SLNode** class. It has an **Object** as a data field that, in this case, is protected instead of private as it was in the **SLNode.** The data **Object** must be declared as protected to allow the subclasses access to the data field in the superclass. All the public accessor methods for the data field are the same as in the **SLNode.** The default **equals()** method is retained as well to allow for data field comparisons.

The next class that has been rewritten to provide a generic superclass is the **SLEnumeration** class (see Figure 6-2). Its purpose is to provide the traversal functionality in the list. The new superclass is called **ListEnumeration.** Like its predecessor, it implements the **Enumeration** interface from the core Java utility package.

We should note some important changes in the **ListEnumeration** class. The object passed to the constructor in the old **SLEnumeration** class was of type **SLNode.** In the new class, it is now of type **Node,** our new node superclass. We want to use the generic **Node** as the parameter so that all the derived enumeration classes will share a common root. Most of the functionality of the enumeration is supplied by this superclass using the **Node** class. The only thing the subclasses will need to do is implement the **nextElement()** method.

The **nextElement()** method has been declared as being an abstract method. The **abstract** keyword in a Java method declaration indicates that the definition of the method is not provided. This forces the user to create a subclass to define the method with the signature provided. This is similar in function to a Java interface declaration. In an interface, all the declared methods are abstract and require a class to "implement" the interface by defining the method bodies. Trying to instantiate an abstract class directly within a Java application generates a **RunTimeException.** In this case, we declare the method **abstract** because the **nextElement()** method is supposed to follow the link to the "next" element in the list. No link mechanism is defined in the **Node** class, though. We expect **Node** to be subclassed to implement the link functionality, which will determine how the "next" element is reached.

The biggest change is the transformation of the **SLinkedList** class to the **LinkedList** class. It is no longer a class at all. **LinkedList** defines an interface rather than a class. The Java interface is roughly equivalent to a class in

Figure 6-2
ListEnumeration.java.

```
package adt.Chapter06;

import java.util.Enumeration;
```

```
public abstract class ListEnumeration
     implements Enumeration
     {
     public ListEnumeration( Node first )
     {
          if( first == null )
               throw new NullPointerException("List is
               empty");
          current = first;
     }

     public boolean hasMoreElements()
     {
          return (current != null );
     }

     public abstract Object nextElement();

     public void print()
     {
          while( hasMoreElements() )
          {
               System.out.println( nextElement() );
          }
     }

     Node current;
}
```

which all methods are declared to be **`abstract`**. The big differences between an interface and an **`abstract`** class revolve around inheritance. An **`abstract`** class, like all classes in Java, is a subclass of **`Object`**. It inherits all the members of **`Object`**, as would any other subclass. An interface, on the other hand, is not a class and so does not inherit any of the properties of the **`Object`** class. The inheritance limit of one direct superclass for each class does not apply, because an interface merely lists required methods that are present in the class. A class is allowed to implement as many interfaces as desired.

In this case, the interface defines the minimum functionality that a linked list must provide. The interface also defines the signature for each of the required methods. By using Java interfaces, the developer ensures compliance with a particular standard API. The **`LinkedList`** interface (see Figure 6-3) defines a minimally required seven specific methods that must be implemented to conform with the linked list specification.

Figure 6-3
LinkedList.java.

```
package adt.Chapter06;

public interface LinkedList
{
     public void add( Object o );
     public void insert( Object o );
```

```
     public void delete();
     public void reset();
     public void setCurrent( Object o );
     public Object getCurrent();
     public ListEnumeration elements();
}
```

The most important thing we should keep in mind here is that we don't expect the **`Node`**, **`ListEnumeration`**, or **`LinkedList`** to be used alone. As a matter of fact, because they are abstract, the **`LinkedList`** and **`ListEnumeration`** are not able to be used directly. An **`abstract`** class must be subclassed to be used at all.

`Node`, **`ListEnumeration`**, and **`LinkedList`** comprise the basic building blocks we'll use to create the linked list derivatives in this and the following chapters. Now that we have this foundation, take a look at the first of the linked list derivatives: the doubly-linked list.

A Doubly-Linked List

In Chapter 5, we looked at the singly linked list. Although these lists are quite useful under certain circumstances, singly-linked lists do have disadvantages. Every access must start from the head of the list, and movement through the list is unidirectional. This makes some of the operations, such as searching and appending on the list, less convenient than they could be otherwise. The doubly-linked list is a variation of the linked list that doesn't have these disadvantages. It provides bi-directional links in the list. These links make it possible to traverse the list in either direction—forward or backward.

In Chapter 5, we learned that, when working with the **`SLinkedList`** class, that the **`delete()`** method needs to identify the node in the list immediately preceding the node to be deleted. The list has to be traversed from the beginning each time to find the node that precedes the node to be deleted. The preceding node is required so that we can save the **`next`** reference from the deleted node to restore the chain. Traversing the list from the beginning to get a reference to the preceding node can be

Figure 6-4
DLNode.java.

```
package adt.Chapter06;

class DLNode
     extends Node
{
     DLNode()
     {
          super();
          next = prev = null;
     }

     DLNode( Object data )
     {
          super(data);
          next = prev = null;
```

```
    }

    void setNext( DLNode next )
    {
        this.next = next;
    }

    DLNode getNext()
    {
        return next;
    }

    void setPrev( DLNode prev )
    {
        this.prev = prev;
    }

    DLNode getPrev()
    {
        return prev;
    }

    private DLNode next;
    private DLNode prev;
}
```

avoided in a doubly-linked list, beause each node can directly identify the preceding node as well as the following one.

We will create the **Node** class for a doubly-linked list, **DLNode**, by subclassing our new **Node** class (see Figure 6-4). The data field and all the accessor methods of **DLNode** are available from the **Node** superclass. In addition, **DLNode** implements the link functionality for both forward and backward links. This is accomplished by defining two **DLNode** references **prev** and **next**, as well as their accompanying accessor methods.

The **setNext()** and **getNext()** accessor methods maintain the forward link to the node. The **setPrev()** and **getPrev()** accessor methods maintain the new backward link. We now have the mechanism by which we can walk forward or backward through the linked list.

Now take a look at the doubly-linked list class in Figure 6-5.

Figure 6-5
DLinkedList.java.

```
package adt.Chapter06;

import java.util.Enumeration;

public class DLinkedList
    implements LinkedList
{
    public DLinkedList()
    {
        head = null;
```

```
}

public void add( Object o )
{
    DLNode newNode = new DLNode(o);

    if( head == null )
    {
        current = head = tail = newNode;
        return;
    }

    current = tail;
    current.setNext( newNode );
    newNode.setPrev( current );
    tail = current = newNode;
}

public void insert( Object o )
{
    DLNode newNode = new DLNode(o);
    DLNode prev;

    if( head == null )
    {
        current = head = tail = newNode;
        return;
    }

    prev = current.getPrev();
```

Continues

Figure 6-5
Continued.

```
    newNode.setNext( current );
    newNode.setPrev( prev );

    if( prev != null )
        prev.setNext( newNode );

    current.setPrev( newNode );

    if( current == head )
        head = newNode;

    current = newNode;
}

public void delete()
{
    DLNode prev;
    DLNode next;
```

```
        if( current == null )
            return;

        if( current == head )
        {
            current = head = head.getNext();
            head.setPrev( null );
            return;
        }

        if( current == tail )
        {
            current = tail = tail.getPrev();
            tail.setNext( null );
            return;
        }

        prev = current.getPrev();
        current = next = current.getNext();

        prev.setNext(next);
        next.setPrev(prev);
    }

    public void setCurrent( Object o )
    {
        current.setData(o);
    }

    public Object getCurrent()
    {
        return current.getData();
```

Continues

Figure 6-5
Continued.

```
    }

    public void reset()
    {
        current = head;
    }

    public boolean next()
    {
        current = current.getNext();
        if( current == null )
            return false;
        else
            return true;
    }

    public boolean prev()
```

```
    {
        current = current.getPrev();
        if( current == null )
            return false;
        else
            return true;
    }

    public boolean tail()
    {
        current = tail;
        if( current == null )
            return false;
        else
            return true;
    }

    public boolean head()
    {
        current = head;
        if( current == null )
            return false;
        else
            return true;
    }

    public DLEnumeration elements()
    {
        return new DLEnumeration( head );
    }

    protected DLNode head;
    protected DLNode tail;
    protected DLNode current;
}
```


As with the **DLNode** class, the **DLinkedList** class is very similar to its singly-linked counterpart in the methods and data fields it provides; the **LinkedList** interface determines the names and signatures of the main methods in the class. One notable difference between **DLinkedList** and **SLinkedList** is that, in **DLinkedList**, an additional data field member called **tail** exists. The **DLNode tail** is used in much the same way as the **DLNode** and **SLNode head**; it provides a placeholder that points to the element at the end of the list.

In the singly-linked list, **head** is used as the base point for many of the operations. We must "reset" the list and traverse it from the beginning to locate a particular element of interest. We can use the **tail** field in the **DLinkedList** in much the same fashion. Instead of starting at the beginning of the list and walking forward through it, we can use the **tail** field to start at the end of the list and walk backward through it.

Even though the methods are by and large the same as the **SLInkedList**, the implementation of the methods in the **DLinkedList** is a little different with the addition of the **tail** field.

The **add()** method checks to see whether the node being added is the first node in the list. If it is, the **head, current,** and **tail** are set to point to the new node. If not, the node is added to the end of the list. In the **SLinkedList,** the entire list must be traversed to find the end. In the **DLinkedList**, the **tail** is already there. After the new node is added, the **tail** field must be reset to point to the new end of the list (**newNode**).

The **insert()** method performs the same check as **add()** for an empty list. Otherwise, the new node is inserted before the current node by using the **next** and **prev** fields to reset the links. Again, the need to traverse the list to find the previous node is eliminated. Because we insert before the current node, there is no need to maintain the tail reference as long as the list is not empty The head reference still must be maintained, though, as before. If the current node is the head node, the new node will have to become the new **head.**

The **delete()** method needs to check for two special conditions: whether the node being deleted is the **head** or **tail** node. In either case, the end node is reset to the subsequent or preceding node, respectively. If the node falls under either of these special cases, the delete operation is much simpler than in the singly-linked list. Again, we avoid traversing the list, because we simply need to set the previous and next nodes to point to each other and drop the current node out of the list.

There are no changes at all to the **setCurrent(), getCurrent(), reset(),** and **next()** methods. A few additional methods also are added for convenience. The **prev()** method enables the user to back-step

through the list. The user also can use the **tail()** and **head()** methods to set the **current** reference to either end of the list.

Finally, the **elements()** method is changed only to return a **DLEnumeration** object derived from **ListEnumerator** instead of the **SLEnumeration** object.

Circular Linked lists

In the previous linked list derivatives, the end nodes are indicated by the **next** and/or **prev** reference being set to **null.** (Remember that new nodes are initialized with all of the pointers set to **null.)** The list can be set to one endpoint and traversed until the **null** reference indicates that the other endpoint to the list is reached.

The next derivative of the linked list we'll look at has a slightly different arrangement. Instead of having the first and last nodes in the list reference **null,** they reference each other. This type of linking is known as *circular,* in contrast to the *linear* approach used so far (see Figure 6-6).

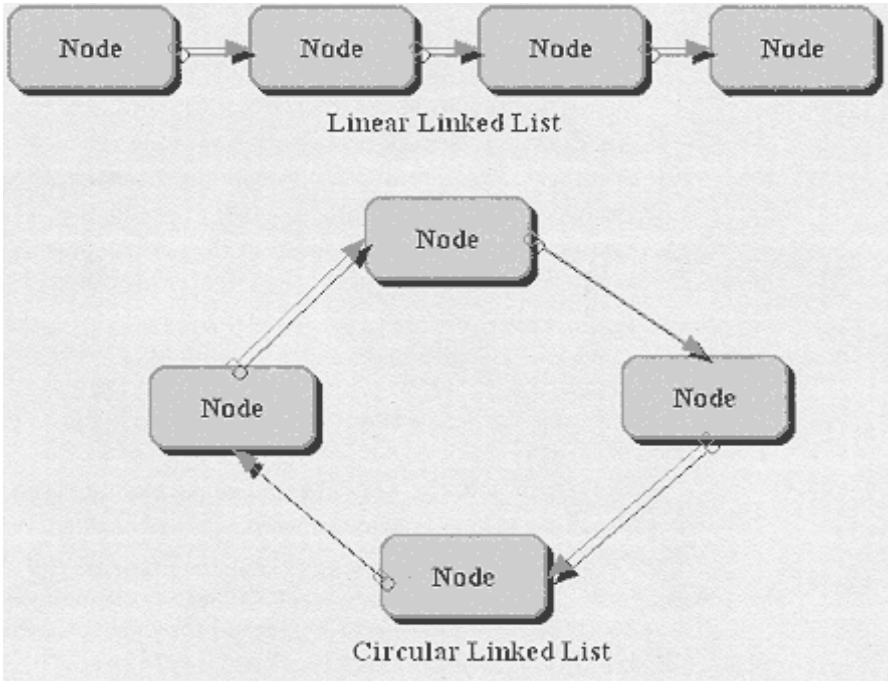


Figure 6-6
Unlike the linear linked list, any node may be the head of a circular linked list without losing access to any of the nodes.

Figure 6-6 illustrates the basic concept of the circular linked list. Looking at the illustration, it is very clear which nodes are the head and tail of the linear linked list. It is not clear which node is the head of the circular list. This raises an interesting question: Is the concept of a head and tail node appropriate for a circular linked list? The answer depends on the intended use of the list.

One reason to opt for a circular design is that it can simplify the implementation. Maintaining the two-way links in a linear implementation generally involves separate references for the head and tail of the list. In a circular list, the tail easily can be computed from the head node (**`head.getPrev()`**). This removes the need for a separate reference and the code needed to maintain that reference. This approach offers little in the way of additional functionality, but it can improve the performance of the list.

A different reason for using the circular design is the capability to have a floating head to the list. Because the list is circular, there is no underlying requirement that the head of the list is any particular node. The **`current`** reference used in the examples so far in this chapter easily could be considered the head of the list for any particular operation. Operations that traverse the list just need to continue through one complete cycle at the maximum, and possibly less if the operation is searching for a particular node in the list.

The problem with this approach is that it makes it very difficult to maintain a specific order to the list. Imagine traversing a list to find the correct place in which to insert a new node. Because the nodes are inserted into the list in order, the usual process is to traverse the list until the current node compares larger than the node being inserted. But with a floating head node, at some arbitrary point in the traversal, the value of the nodes you are comparing to

drops. This requires the comparison method not only to compare the current node with the target node but also with the previous node to determine whether the node with the maximum value has been reached.

For this example, we will implement a hybrid implementation. We'll keep track of the true head of the list and process as normal for most operations. We'll also allow traversal from an arbitrary point if desired. This "floating head" behavior will be restricted to the list enumeration.

First look at the new **`ListEnumeration`** subclass in Figure 6-7. As in the previous **`ListEnumeration`** classes, we need to pass the constructor a reference to the head of the list. In this case, though, we need to retain this value to check whether we've cycled through the whole list. In our previous implementations, we checked for a **`null`** reference to indicate the end condition. In a circular list, this will not occur.

Figure 6-7
CLEnumeration.java.

```
package adt.Chapter06;

import java.util.Enumeration;

public class CLEnumeration
    extends ListEnumeration
{
    public CLEnumeration( DLNode first )
    {
        super(first);
        if( first == null )
            throw new NullPointerException("List is
empty");
        current = first;
        this.first = first;
    }

    public boolean hasMoreElements()
    {
        if( super.hasMoreElements() == false ||
             ((DLNode)current).getNext() == first )
            return false;
        else
            return true;
    }

    public Object nextElement()
    {
        Object o = current.getData();
        current = ((DLNode)current).getNext();
```

The **`first`** field will hold a reference to the head of the list. As far as our enumeration class is concerned, this could be the real head of the list, or it could be the "floating head" that has been discussed. To the enumeration, there is no difference as to which node it uses as a starting

point. This enumeration is designed to loop through the entire list once from the starting point indicated. Where that starting point is in comparison to the "real" head of the list makes no difference and causes no change in processing or procedure.

For this subclass of **`ListEnumeration`**, all the methods have to be defined. We can't do with the default **`hasMoreElements()`** method from the superclass, because it doesn't process a circular list. In this version of **`hasMoreElements()`**, the superclass version of the method is called first. This ensures that the error checking for an empty list is done before our check for a "wrapped" entry is performed. You might be asking, "Why not just

do the same check here as is done in the superclass and save the extra method call?" That's a good question. As the classes currently stand, it would be easy to incorporate the error-checking code into our new method and skip the call to **`super`**. But, once again, we need to think about extensibility. What if the nature of the list changes so that other validation tests are required for the enumeration? By incorporating a call to **`super.hasMoreElements()`**, we ensure that changes of this sort are included automatically in our subclass without us having to re-edit and recompile. Now, take a look at the **`CLinkedList`** class shown in Figure 6-8.

The big difference between the **`DLinkedList`** and **`CLinkedList`** is that the tail reference is gone and, in its place, the head reference points to the "end" of the list as its previous node.

Figure 6-8
CLinkedList.java.

```
package adt.Chapter06;

import java.util.Enumeration;

public class CLinkedList
    implements LinkedList
{
    public CLinkedList()
    {
        head = null;
    }

    public void add( Object o )
    {
        DLNode newNode = new DLNode(o);
        DLNode prev;

        if( head == null )
        {
            current = head = newNode;

            head.setNext( current );
            head.setPrev( current );
            return;
        }

        prev = head.getPrev();
```

```
        prev.setNext( newNode );
        head.setPrev( newNode );

        newNode.setNext( head );
        newNode.setPrev( prev );

        current = newNode;
```


Figure 6-8
CLinkedList.java.

```
    }

    public void insert( Object o )
    {
        if( head == null )
        {
            add(o);
            return;
        }
         DLNode newNode = new DLNode(o);
         DLNode prev;

         prev = current.getPrev();

         newNode.setNext( current );
         newNode.setPrev( prev );

         prev.setNext( newNode );
         current.setPrev( newNode );

         if( current == head )
         head = newNode;

         current = newNode;
    }

    public void delete()
    {
        DLNode prev;
        DLNode next;
        DLNode tail;

        if( current == null )
            return;

        tail = head.getPrev();

        if( head == tail )
        {
            current = head = null;
            return;
        }

        if( current == head )
```

```
        {
            current = head = head.getNext();

            head.setPrev( tail );
            tail.setNext( head );
            return;
        }
```

Continues

Figure 6-8
Continued

```
        if( current == tail )
        {
            current = tail = tail.getPrev();

            tail.setNext( head );
            head.setPrev( tail );
            return;
        }

        prev = current.getPrev();
        current = next = current.getNext();

        prev.setNext(next);
        next.setPrev(prev);
    }

    public void setCurrent( Object o )
    {
        current.setData(o);
    }

    public Object getCurrent()
    {
        return current.getData();
    }

    public void reset()
    {
        current = head;
    }
    public boolean next()
    {
        current = current.getNext();
        if( current == null )
            return false;
        else
            return true;
    }

    public boolean prev()
    {
        current = current.getPrev();
```

```
        if( current == null )
            return false;
        else
            return true;
    }

    public boolean tail()
    {
        current = head.getPrev();
        if( current == null )
            return false;
        else
            return true;
    }

    public boolean head()
    {
        current = head.getPrev();
        if( current == null )
```

Continues

Figure 6-8
Continued.

```
            return false;
        else
            return true;
    }

    public boolean head()
    {
        current = head;
        if( current == null )
            return false;
        else
            return true;
    }

    public ListEnumeration elements()
    {
         return new DLEnemeration(head );
    }

    public ListEnumeration elements( boolean fromCurrent)
    {
        if( fromCurrent )
            return new DLEnumeration( head );
        else
            return new CLEnumeration( head );
    }
    protected DLNode head;
    protected DLNode current;
}
```

Performance Considerations

A big advantage of both these types of linked lists is the increase in speed brought about by the elimination of the need to traverse the list on insert and delete operations in the middle of the list. In a list that primarily appends or prepends to the list, there is no big advantage to a doubly or circularly linked list. The same rule applies to a list in which deletes are performed primarily to the first and last nodes in the list. As a matter of fact, the additional overhead of maintaining the second set of references actually degrades the performance of the list in these situations.

For these bi-directionally linked list types, we've abandoned the vector approach to the implementation. This is not to say that a vector solution is not viable. In the context of performance, though, the vector has the most overhead in the insert and delete operations—the very same operations that make the bi-directionally linked lists attractive.

Exercises

1. Implement the **`DLinkedList`** class by using the vector approach and compare its performance to the reference implementation presented in this chapter.

2. Use the quicksort algorithm to create a linked list class that automatically orders a set of comparable elements.

Summary

In this chapter, we learned the following:

- We defined new superclasses for the node and list enumeration to simplify further extensions of the linked list.
- We created a new interface to define the properties of a linked list type. This interface supports easier extension of the linked list.
- We examined the concept of a bi-directionally linked list and compared it to the unidirectional list covered earlier.
- We looked at circular linking and the advantages it offers.
- We examined some guidelines to help us decide which type of linked list to use in different circumstances.

Chapter 7
Stacks

This chapter takes a look at the stack as a specialized linked list. The built-in Java **`Stack`** object is used as an example of a vector-based stack. An analysis of the internals for the stack is provided, and a non-vector implementation is developed as a contrast. Exercises near the end of this chapter give us an opportunity to look at the uses of the stack.

A Specialized Linked List—the Stack

In Chapter 1, "Basic Concepts," we saw a data construct called a *stack.* It was used to describe the mechanism the Java platform uses when passing arguments to a method when it is called. A stack is another of the abstract data types based on the linked list. More specifically, a stack is derived from the singly-linked list.

In Chapter 6, we learned that singly-linked lists are best used in situations in which most of the additions to and deletions from the list will occur at one of the endpoints. The stack definitely meets this criterion. *A stack* is a data storage mechanism of a type known as *last in, first out* (LIFO). A LIFO type has only two basic operations it is responsible for: **`push`** and **`pop`**.

These operations are roughly analogous to the **`put`** and **`get`** operations used with the hash table, as discussed in Chapter 4, "Hash Tables." Remember that the **`put`** operation adds a data item to the hash table in a position determined by the hash table rather than the user. An internal algorithm determines where in the table the element belongs. A **`get`** operation retrieves an element based on the same algorithm.

`Push` and **`pop`** are very similar to **`put`** and **`get;`** they are used to store and retrieve data from the stack. The difference is in the algorithm used to determine the storage position. In a LIFO, the add operation is always performed at the front (also called the *top* or *head* of the stack). Each add therefore *pushes* the node before it deeper into the list. By the same token, each get operation (deletion) moves a data item off the top of the stack (or *pops* it off the stack). This operation also moves the rest of the data in the stack closer to the top by one position. Figure 7-1 illustrates the **`push`** and **`pop`** operations.

To better visualize the mechanics of a stack, think about the spring-loaded plate servers at most restaurant salad bars. Clean plates are loaded onto a spring-loaded platform for the customers to use. Imagine an employee loading the plates onto the platform one at a time. As each plate is added, it *pushes* the plates before it deeper into the stack. Customers have easy access only to the top plate in the stack. As each customer comes by and *pops* a plate off the top, each of the remaining plates are moved one plate closer to the top. In other words, the last plate added to the stack is always the first plate removed from the stack—last in, first out.

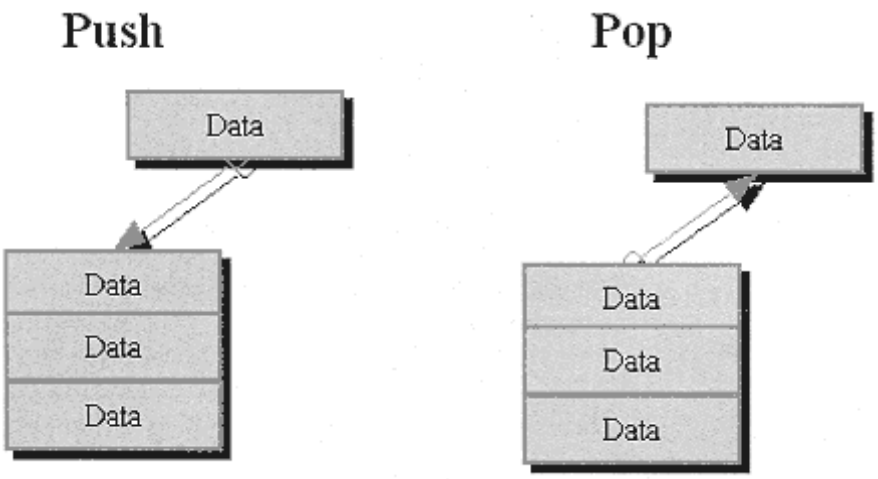


Figure 7-1
Pushing a node onto a stack and popping a node off of a stack.

The Java Core Class: `java.util.Stack`

Among the core classes provided by the Java Developer's Kit is an implementation of a stack type called, appropriately, **`Stack`**. The Java **`Stack`** is extended from the **`vector`** class, making it an array-based implementation. Earlier, we learned that the most expensive operations in a vector, in terms of time, are the operations that require a part or all of the underlying array to be copied. The Java implementation of the stack makes sure to minimize the need to copy.

Remember that no insert operation exists on a stack—just **`push`** and **`pop`** (**`add`** and **`get`** from previous examples). Also, no delete operation from the middle of the list exists. All the operations are performed at the end of the list. With this stipulation, the vector can be quite effective as the underlying basis for a stack.

The Java **`Stack`** class is a subclass of **`Vector`**. This means that all the public methods in **`Vector`** are available in **`Stack`**. Using the standard **`Vector`** methods, such as **`insertElementAt()`** or **`removeElement()`**, defeats the purpose of using a stack in the first place and so should be avoided. With that in mind, this section covers only the methods provided directly in the **`Stack`** class.

The **`push()`** method adds an item to the top of the stack:

```
public Object push(Object item)
```


In the case of the vector implementation of the stack, the top of the list is considered to be the last element in the underlying array. The Java implementation uses the vector's **`addElement()`** method to append the item to the list.

The **`pop()`** method extracts the top item in the stack:

```
public synchronized Object pop()
```

In the vector implementation, the top of the stack is the element at the index **`Vector.size() - 1`**. Of course, the actual index of the top element changes with each **`push`** and **`pop`**. The fact that the **`Stack`** always works at the end of the vector instead of the beginning means that the number of times an array copy occurs is limited to the number of times the size of the underlying array must be expanded.

The **`peek()`** method provides additional functionality that can be quite handy:

```
public synchronized Object peek()
```

`Peek` enables the user to view the element at the top of the stack without actually removing the item from the stack. Think of **`peek`** as a nondestructive **`pop`**. The **`peek`** operation is not strictly necessary to the stack, but it is a nice feature.

The **`empty()`** method is used to determine whether the stack is empty:

```
Public boolean empty()
```

If the **`Vector.size() == 0`**, the stack is empty. Again, this method is not necessary to the stack but certainly is useful.

The **`search()`** method is another addition to the **`Stack`** class that is not strictly necessary but is a nice feature:

```
public synchronized int search(Object o)
```

The method returns the distance from the top that the supplied **`Object`** is located. If the **`Object`** is not found in the stack, -1 is returned. This implementation uses the **`Vector.lastIndexOf()`** method to determine the position in the array and then subtracts that index from the number of elements in the array **`(vector.size())`** to determine how far the element is away from the top of the stack.

We also should know that, although **`Stack`** is a subclass of **`Vector`**, it provides only the default constructor. The user of the **`Stack`** class cannot specify the initial size of the underlying array or the capacity increment of the **`Stack`** (as was possible with the **`Vector`**'s constructors). The default

constructor provides an initial size of 10. The default action for resizing the array is to double it each time it becomes full.

Uses of the Stack

The stack is used in many cases when the desired action is to access the data in the reverse order in which it was stored. This is similar to the undo function in word processing. Keystrokes and the actions caused by them can be undone by unapplying them in reverse order.

Refer to the example in Chapter 1 of the mechanism used to pass arguments to a method when it is called. The calling method first pushes its current instruction address onto the stack, and then the arguments are pushed onto the stack. After program control is passed to the method, the arguments are popped back off the stack. And, finally, when the method is ready to return, it

pops the return address off the stack and passes control back to the calling method.

The stack also is very useful when quick data storage and access are required and order is not necessarily important. No traverse operation is necessary to a stack, because all access to data is through a single point. No overhead is involved in looking up a data item, because the next data item is the one at the top of the stack.

A Reference-Based Stack

Because we've already looked at the Java core class vector-based stack, it's time to take the next step and define a reference-based stack class type of our own. It is possible to create a stack type with just a few lines of Java code, as shown in Figure 7-2.

By looking at the **`SimpleStack`** class in Figure 7-2, it is obvious that a reference-based stack implementation is extremely simple. We've included only **`push()`** and **`pop()`** methods, with only a default constructor.

The mechanics of the reference-based stack are as basic as the class itself Like all our linked list implementations, the **`SimpleStack`** uses a node class to store the data. In this case, we define a package private **`SimpleStackNode`** to do the job. The **`SimpleStackNode`** class defines no methods. It has only the two package private (default access) fields in which to store a reference to the data object and the next node down in the stack.

Figure 7-2
SimpleStack.java.

```
package adt.Chapter07;

public class SimpleStack
{
    public void push( Object o )
    {
        SimpleStackNode tmp = new SimpleStackNode();
        tmp.data = o;
        tmp.next = top;
        top = tmp;
    }

    public Object pop()
    {
        if( top == null )
            return null;

        SimpleStackNode tmp = top;

        top = top.next;
        return tmp.data;
    }
    SimpleStackNode top;
}

class SimpleStackNode
```

```
{
    Object data;
    SimpleStackNode next;
}
```

As a general rule of thumb, defining a class with exposed (public) data members is a bad idea. There is very poor data encapsulation for the member fields, because they are visible and accessible to any class in the package. Breaking encapsulation like this also severely limits the extensibility of the whole stack construct by making the **SimpleStack** class explicitly dependent on the **SimpleStackNode** implementation. Any change to the data types of the **SimpleStackNode** member fields most likely will cause a need for the **SimpleStack** class to be modified accordingly.

Why was the **SimpleStackNode** implemented this way? To provide an example of one instance in which this kind of class can be used effectively. Notice that the **SimpleStackNode** class is defined in the same source file as **SimpleStack** Java allows multiple classes to be defined in a single source file, as long as only one of the classes is defined as public. This practice also is frowned upon, but it is allowed for precisely this kind of situation. We should define multiple classes in the same source file if the package pri-

vate class is used by the **public** class only If a class outside the source file containing the package private class (with the **public** class) uses or extends the package private class, it should be defined in its own source file.

Now take a look at exactly what **push()** and **pop()** are doing. The **push()** method creates a new node and populates the **data** field. The actual *push* part of the operation is performed by setting the new nodes **next** field to point to the previous top of the stack. This effectively maintains the chain of nodes without the need to traverse the list. The final step to the push is to reassign **top** to refer to the newly created node.

The **pop()** method is almost as easy. First, we need to check that the list is not empty. If **top** is **null**, the list is empty, and **null** is returned. Otherwise, we can proceed with the **pop** operation. Because we are going to remove the top node from the list, but we don't want to lose the data associated with it, we need to keep a temporary reference to the node. This is done to protect against the node being prematurely garbage collected and because, after we reassign **top**, we lose the reference and have no way to get it back. This is a singly-linked list, remember. The local variable **tmp** therefore is assigned with the reference to **top.** All that remains is to reassign **top** to the **next** node in the list and to return the **data** object.

That is all there is to creating an extremely simplified but fully functional stack type. Figure 7-3 shows a minimal demonstration program for the **SimpleStack** class, and Figure 7-4 shows the output from the program.

Figure 7-3
SimpleStackTest.java.

```
package adt.Chapter07;

public class SimpleStackTest
{
```

```
    public static void main( String args[] )
    {
        SimpleStack ss = new SimpleStack();
        String s;

        ss.push( "this" );
        ss.push( "comes" );
        ss.push( "out" );
        ss.push( "in" );
        ss.push( "reverse" );
        ss.push( "order" );

        while( (s = (String)ss.pop()) != null )
            System.out.println( s );
    }
}
```


Figure 7-4
Output from the **SimpleStack** class.

```
order
reverse
in
out
comes
this
```

The **SimpleStackTest** program pushes the word strings onto the stack one at a time and pops and displays each word as it comes off the stack. Notice that the defining feature of the stack is that the output is in the reverse order of the input.

SimpleStack was quick and easy to implement, but it is probably not appropriate for large-scale development because the stack construct is so dependent on the implementation details of its node class. It would be much more desirable to extend the more robust superclasses we defined previously to allow for easy extensibility and maintenance than would be available in our **SimpleStack** example.

To come up with a more robust implementation, we need to start by defining the node class for the stack. Once again, we will extend our generic **Node** class and add the **next** reference, as shown in Figure 7-5.

The **StackNode** subclass defines two constructors to match the two in the superclass. The only additional work done here is to initialize the **next** reference to **null**. Then the class supplies accessor methods for the next reference. We now have a much safer node implementation with very little additional effort.

The next task is to define the **Stack** class itself (see Figure 7-6). Of course, we will be implementing the **LinkedList** interface we defined in Chapter 6.

In the source listing, notice that the class defines **push()** and **pop()** methods that are almost identical to the ones in **SimpleStack**. The only difference is that the **StackNode**'s **next** reference is manipulated through the accessor methods provided by the **StackNode** class.

The remaining methods (the ones from the interface) are very sparsely coded. One of the reasons for defining the interface in the first place is to allow the different **`LinkedList`** types to be used interchangeably. Unlike the vector-based Java core **`Stack`** class, though, we don't want to allow the user to break the order of the stack. To accomplish both of these goals, we can force the **`add()`**, **`insert()`**, and **`delete()`** methods to use **`push()`** and **`pop()`** to perform their operations.

This does change the behavior of the methods from that of the other implementations, but it is consistent with the expected behavior of a

Figure 7-5
StackNode.java.

```
package adt.Chapter07;

class StackNode
    extends Node
{
    StackNode(Object o)
    {
        super(o);
        next = null;
    }

    StackNode()
    {
        super();
        next = null;
    }

    StackNode getNext()
    {
        return next;
    }

    void setNext(StackNode next)
    {
        this.next = next;
    }

    private StackNode next;
}
```

Figure 7-6
Rstack.java.

```
package adt.Chapter07;

import adt.Chapter06.LinkedList;
import adt.Chapter06.ListEnumeration;

public class RStack
    implements LinkedList
```

```
{
    public RStack()
    {
        top = null;
    }

    public void push(Object o)
    {
        StackNode tmp = new StackNode(o);
        tmp.setNext(top);
        top = tmp;
```

Continues

Figure 7-6
Continued.

```
    }

    public Object pop()
    {
        StackNode tmp = top;
        top = top.getNext();
        return tmp.getData();
    }

    public void add(Object o)
    {
        push(o);
    }

    public void insert(Object o)
    {
        push(o);
    }

    public void delete()
    {
        pop();
    }

    public void reset()
    {
        ;
    }

    public void setCurrent(Object o)
    {
        ;
    }

    public Object getCurrent()
    {
        return top.getData();
    }
```

```
    public ListEnumeration elements()
    {
        return null;
    }

    StackNode top;
}
```

stack. For example, **`push`** is analogous to **`add`** in the stack vernacular. It is not unreasonable to expect the **`add()`** and **`push()`** methods to perform exactly the same operations.

The insert operation is not really appropriate to a stack at all. We defined that all additions to the stack are performed at the top of the stack. *Insert* implies that the new element added will be somewhere in the middle of the list. So one option for the **`insert()`** method is to have it do nothing at all, because the stack doesn't allow traditional insertions. Some precedence exists, however, for an insertion to be performed at the list endpoint. Remember that an insert to an empty list performs an add operation. Although this might be a bit of a stretch as a justification, it is better to have the **`insert()`** method do a **`push`** and follow the conventions of the stack than to have it do nothing at all.

The **`delete()`** method always has been used so far to remove the current element in the list. In the case of the stack, the current element is always the top element in the stack. So it makes perfect sense to have the **`delete()`** method call the **`pop()`** method. The difference is that the **`pop()`** method is defined to return the reference to the "popped" element. The **`delete()`** method is not defined to return anything, so the reference is dropped.

To counter the fact that the **`delete()`** method pops an element off the list without returning a reference, the **`getCurrent()`** method returns a reference to the top element in the stack without performing a pop. This operation is similar to the **`peek()`** method provided by the Java core class **`Stack.`**

The **`reset()`** and **`setCurrent()`** methods aren't appropriate at all for the stack. The purpose of the **`reset()`** method originally was to force the node at the endpoint (usually, **`head`**) to be the current node. With the stack, the current node is always the **`top`** node. The reset operation therefore is meaningless to the stack and doesn't do anything. We do have to define the method, though, or the **`LinkedList`** interface will not be completely implemented. So we define it as an empty method.

The same basic argument applies to the **`setCurrent()`** method. The difference, of course, is that **`setCurrent()`** was expected to find a particular element in the list and make the corresponding node the current node in the list. The only way to make a particular element current in the stack is to pop off all the elements until the desired one is found. That functionality is very counterintuitive, so it is best to leave this method empty as well.

The final method defined in the interface is the enumerator, **`elements().`** The enumeration of our linked lists so far have been non-destructive. There is no overwhelming reason to change that behavior with the stack, so the **`elements()`** method gives the user of the stack a way to peruse the contents of the stack without affecting the state of the stack.

Listing 7-7
StackEnumeration.java.

```
package adt.Chapter07;

import adt.Chapter06.ListEnumeration;

public class StackEnumeration
    extends ListEnumeration
{
    public StackEnumeration( StackNode first )
    {
        super(first);
        if( first == null )
            throw new NullPointerException("Stack is
empty");
        current = first;
    }

    public Object nextElement()
    {
        Object o = current.getData();
        current = ((StackNode)current).getNext();
        return o;
    }
}
```

This brings us to the last class needed to complete our stack implementation: **`StackEnumeration`** (see Figure 7-7). The implementation is almost exactly like the enumeration demonstrated in Chapter 6.

With the **`StackEnumeration`** class, the user has the capability to walk the contents of the stack without having to pop each element off the stack.

Exercises

1. Construct an application that uses the Java core class **`Stack`** to store a list of words. Display the stack by using the **`pop()`** method. Substitute the **`Vector`** class for the **`Stack`** class in the same application. Use the appropriate **`Vector`** methods to perform the **`push`** and **`pop`** operations.

2. Construct an application using the Java core class **`Stack`.** Use the superclass **`Vector`** **`insertElementAt()`** method to store a list of words alphabetically in the stack (even though this makes no sense in the context of the stack). Then use the **`Stack pop()`** method to display the contents of the stack.

3. Construct an application to use the **`Random`** class to generate a list of 10,000 four-character strings. Compare the performance of the **`RStack`** and the **`Stack`** to push and pop these 10,000

strings.

Summary

In this chapter, we learned the following:

- Stacks are specific implementations of singly-linked lists.
- Stacks are last in, first out (LIFO) constructs.
- Stacks are somewhat similar to hash tables.
- We can implement stack functionality in several ways.

Chapter 8
Queues

This chapter explores another specialization of the linked list: the queue. Queues are used in systems that require message handling, event processing, and the sharing of resources such as printers. Throughout the chapter, we'll walk through the concepts behind, and the implementation of, a standard *first-in/first-out* (FIFO) queue. We'll compare the queue storage container to the stacks covered in the preceding chapter and their *last in, first out* (LIFO) schema. Once again, we'll take a look at both vector and non-vector implementations of the queue in the examples and exercises provided in this chapter.

The FIFO Queue

When we discussed the stack type in the preceding chapter, we determined that it used a *last in, first out* (LIFO) schema. The queue type we'll examine in this chapter is of the *first in, first out* (FIFO) variety, which means that the elements of the list will be accessed in the exact order in which they were added to the list.

The queue is analogous to a grocery checkout line or a waiting line at a bank. The customers are lined up and handled in the order in which they arrived. If 10 people are ready to check out their groceries or to see a teller, they line up behind each other as they come in at one end of the line and are processed from the other end of the line. The first person in line is checked out (processed) first, and the tenth person in line is checked out tenth.

Like the stack type with its **`push`** and **`pop`** operations, only two basic operations are required on a queue: **`put`** and **`get`** (see Figure 8-1). The **`put`** operation is expected to add an element to the end of the list, and the **`get`** operation is expected to extract the first item in the list.

Notice that no operations are defined for manipulating elements in the middle of the list.

As was discussed earlier, the queue is another of the linked list derivatives appropriate for implementation as a singly-linked list. In Chapter 6, we determined that the singly-linked list is best used when the majority of the operations on a list is performed at the endpoints of the list. The queue meets this criterion handily.

Queue Versus Stack

The **`put`** and **`get`** operations for the queue occur at opposite ends of the list, unlike the stack, in which all operations are performed on a single endpoint. In the queue, all the **`put`** operations are performed at the tail end of the list, whereas all the **`get`** operations are performed at the head of the list.

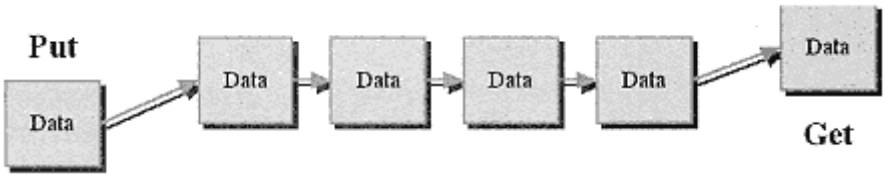


Figure 8-1
All of the operations on a queue are performed on the endpoints of the list.

A Vector-Based Queue

The Java core classes do not include an implementation of the queue. Unlike the **`Stack`** class, the queue is not as efficiently rendered as an extension to the vector It is not that it is more difficult to implement the queue class as a vector extension, but that the performance limitations of the vector are emphasized when using the queue schema.

As pointed out in Chapter 3, "Arrays, Vectors, and Sorting," a big performance hit associated with using a vector comes from the need to perform an array copy when the underlying array needs to grow or when an element is inserted or deleted within the collection. Unfortunately, implementing a vector-based queue tends to do a lot of array copying. If the implementation "gets" elements from the end of the vector, it must "put" elements at the first index in the vector (index zero). Every **`put`** after the first element causes the array to be copied (refer to the vector's insert operation in Chapter 3). Alternatively, the **`put`** operation could add elements to the end of the vector. Then the **`get`** operation would cause a deletion of the first node in the vector after retrieving the data element. Deleting the first element in the vector would cause an array copy every time (refer to the vector's delete operation in Chapter 3).

Now we will do a quick implementation of a vector-based queue. In this implementation, we choose to take the performance hit in the **`put()`** method. The reason for choosing **`put`** instead of **`get`** is that the **`put`** operation will cause the vector to allocate more memory for the underlying array when it fills up. Because growing the underlying array will cause the array copy anyway (refer to the add operation in Chapter 3), we will isolate the performance issues in this one method.

The actual class implementation is very straightforward. Figure 8-2 shows the source code

listing for our vector-based queue class.

The entire **VQueue** class consists of two methods: **put()** and **get()**. No constructor is supplied, so the class uses a default constructor. Any class that doesn't define its own constructor uses a constructor by default, which is equivalent to the following:

```
public Classname()
{
    super();
}
```

Thus, the no-argument constructor for the parent **Vector** class is called when a **VQueue** object is instantiated. Because the **VQueue** class doesn't supply a constructor, there is no way for the user to specify the initial or in-

Figure 8-2
VQueue.java.

```
package adt.Chapter08;

import java.util.Vector;

public class VQueue
    extends Vector
{
    public void put( Object o )
    {
        insertElementAt(o,0);
    }

    public Object get()
    {
        int n = size() - 1;
        Object o = elementAt(n);
        removeElementAt(n);
        return o;
    }
}
```

crement size of the vector. Therefore, the default values are used, which cause the underlying array to be doubled in size each time it is resized.

The **put()** method performs an insert at the beginning of the vector. As mentioned earlier, this causes the vector to perform an array copy each time an element is added. Also, an array copy is executed each time the vector resizes the underlying array. Because we chose to do the additions at the front of the vector, we have effectively isolated all of the resize operations to this one method.

The **get()** method retrieves the element from the other end of the vector. This preserves our FIFO schema. The **get** operation in this implementation never causes the vector's underlying array to be resized.

Consider the following scenario. An element is added to an empty **VQueue** using the **put()**

method, placing it at index zero of the underlying vector. Another element is added to the queue in the same fashion. Our original element is pushed out to index 1, and the new element is added at index 0. Now there are two elements in the vector.

To extract an element, the **`get()`** method is used. We expect to retrieve the original element, because it was the first one added to the queue. To access the original element using the vector's **`elementAt()`** method, the index to extract is determined by subtracting 1 from the size of the vector In this case, the size is 2, so the index to retrieve is 1. After we have a reference to the element, we can remove it from the queue and return the

Figure 8-3
VQueueTest.java.

```
package adt.Chapter08;

public class VQueueTest
{
    public static void main( String args[] )
    {
        VQueue sq = new VQueue();
        String s;

        sq.put( "this" );
        sq.put( "comes" );
        sq.put( "out" );
        sq.put( "in" );
        sq.put( "the" );
        sq.put( "order" );
        sq.put( "as" );
        sq.put( "submitted" );

        while( (s = (String)sq.get()) != null )
            System.out.println( s );
    }
}
```

extracted element to the method caller. The second element then can be removed in the same manner, leaving us once again with an empty queue.

Just as with the Java core class vector-based **`Stack`** class, the user of the **`VQueue`** has access to all the public methods of the **`Vector`** superclass. The same rule we applied to the **`Stack`** holds true for the **`VQueue`,** though: It is inappropriate to use many of the vector methods on the **`VQueue`** because their use could break the **`VQueue`**'s storage scheme (FIFO).

Now that we have implemented the **`VQueue`** class, we need to create a small application to test it. The **`VQueueTest`** class is used for this purpose, as shown in Figure 8-3. The test simply adds a few strings to the queue and then extracts and prints all of them to demonstrate that they retain their order.

A Reference-Based Queue

Because of the performance issues involved with the vector-based queue, it probably is a

better idea to implement the queue as a reference-based linked list. As a first implementation, stick to just the basics and implement only the **`put`** and **`get`** operations, as shown in Figure 8-4.

Figure 8-4
SimpleQueue.java.

```
package adt.Chapter08;

public class SimpleQueue
{
    public SimpleQueue()
    {
        head = tail = null;
    }

    public void put( Object o )
    {
        SimpleQNode tmp = new SimpleQNode();
        tmp.data = o;
        if( head == null )
            head = tmp;
        else
            tail.next = tmp;

        tail = tmp;
    }

    public Object get()
    {
        if( head == null )
            return null;

        Object o = head.data;

        if( head == tail )
            head = tail = null;
        else
            head = head.next;

        return o;
    }

    SimpleQNode head;
    SimpleQNode tail;
}

class SimpleQNode
{
    Object data;
    SimpleQNode next;
}
```

Like all our linked list implementations, the **`SimpleQueue`** uses a **`Node`** class to store the

data. The package private **SimpleQNode** class is used in this case. It has only two member fields: **data** and **next**. No methods are available for this **Node** class. The **SimpleQueue** class also has only the two

package private fields in which to store a reference to the data object and the next node in line in the queue. These fields are both of type **SimpleQNode.** Because this is just a rough implementation (like **SimpleStack** from the preceding chapter), we are not subclassing the **Node** class or implementing the **LinkedList** interface.

The **put()** method is designed to append an element to the list. A new **SimpleQNode** is created and initialized with the data object argument. Next, the new node needs to be put in the list. The **head** is tested for **null** to determine whether the list is empty. If it is empty, **head** and **tail** are both assigned the new node, and the operation is complete. If the list is not empty, the new node is assigned to the **next** reference of the **tail.** The **tail** then is reassigned to the new node, which becomes the new **tail.**

The **get()** method should extract the first element in the list and return it to the caller. To retrieve the node, the list first needs to be checked to see whether it is empty. If the list is empty, the method returns **null.** As an alternative, the **get()** method could throw an exception to indicate that the list is empty, but, in this case, the return of **null** is probably sufficient. If the list is not empty, the data object is extracted from the **head** node. The **head** reference is bumped up one in the queue, and the data **Object** is returned.

This is all that is required in this minimalist implementation of the queue type. To test the **SimpleQueue,** we will modify the **VQueueTest** application to use the **SimpleQueue** class instead of the **VQueue** class. The only change to the original **VQueueTest** required is to change the queue declaration line from

```
VQueue sq = new VQueue();
```

to

```
SimpleQueue sq = new SimpleQueue();
```

Figure 8-5 shows this program listing.

Now that we've looked at a simple reference-based queue implementation, it's time to create an implementation that conforms to the **LinkedList** interface. Of course, we'll need to define a more robust **Node** class than **SimpleQNode. QNode** is subclassed from the **Node** class to provide better data encapsulation (see Figure 8-6).

The next step is to implement the **LinkedList** interface. We therefore define the **Queue** class, as shown in Figure 8-7.

As with the **RStack** class in the preceding chapter, we need to provide implementations of the methods defined by the **LinkedList** interface.

Figure 8-5

SimpleQueueTest.java.

```
package adt.Chapter08;

public class SimpleQueueTest
{
    public static void main( String args[] )
    {
        SimpleQueue sq = new SimpleQueue();
        String s;

        sq.put( "this" );
        sq.put( "comes" );
        sq.put( "out" );
        sq.put( "in" );
        sq.put( "the" );
        sq.put( "order" );
        sq.put( "as" );
        sq.put( "submitted" );
        while( (s = (String)sq.get()) != null )
        System.out.println( s );
    }
}
```

Figure 8-6

Onode.java.

```
package adt.Chapter08;

import adt.Chapter06.Node;

class QNode
    extends Node
{
    QNode(Object o)
    {
        super(o);
        next = null;
    }

    QNode()
    {
        super();
        next = null;
    }

    QNode getNext()
    {
        return next;
    }

    void setNext(QNode next)
    {
        this.next = next;
```

Continues

Figure 8-6
Continued.

```
    }

    private QNode next;
}
```

Figure 8-7
Queue.java.

```
package adt.Chapter08;

import adt.Chapter06.ListEnumeration;
import adt.Chapter06.LinkedList;

public class Queue
    implements LinkedList
{
    public Queue()
    {
        head = tail = null;
    }

    public void put( Object o )
    {
        QNode tmp = new QNode(o);

        if( head == null )
            head = tmp;
        else
            tail.setNext( tmp );

        tail = tmp;
    }
    public Object get()
    {
        if( head == null )
            return null;

        Object o = head.getData();

        if( head == tail )
            head = tail = null;
        else
            head = head.getNext();

        return o;
    }

    public void add(Object o)
    {
        put(o);
    }
```

Continues

Figure 8-7
Continued.

```
    public void insert(Object o)
    {
        put(o);
    }
    public void delete()
    {
        get();
    }

    public void reset()
    {
        ;
    }

    public void setCurrent(Object o)
    {
        ;
    }

    public Object getCurrent()
    {
        return head.getData();
    }

    public ListEnumeration elements()
    {
        return new QEnumeration(head);
    }

    QNode head;
    QNode tail;
}
```

But again, some of the operations in the **`LinkedList`** are not necessarily appropriate for the queue model. In these cases, we force the behavior of the **`LinkedList`** method to match the expected behavior of the queue. We accomplish this by calling the appropriate "normal" queue operation from the method in question.

The **`add()`** and **`insert()`** methods both call the **`put()`** method. The **`add()`** method is equivalent to **`put()`** in this context, so it's easy to see the connection. The **`insert()`** method, on the other hand, normally inserts a new element into the middle of the list. The queue, however, allows operations to take place only at the endpoints of the list. It is not too much of a stretch to let **`insert()`** be the equivalent of **`add()`**, so we have it call put as well.

The next step toward completing this implementation is to define the enumeration class required for the **`elements()`** method (see Figure 8-8).

Figure 8-8
QEnumeration.java.

```
package adt.Chapter08;

import adt.Chapter06.ListEnumeration;

public class QEnumeration
    extends ListEnumeration
{
    public QEnumeration( QNode first )
    {
        super(first);
        if( first == null )
            throw new NullPointerException("Queue is
empty");
        current = first;
    }

    public Object nextElement()
    {
        Object o = current.getData();
        current = ((QNode)current).getNext();
        return o;
    }
}
```

The **`QEnumeration`** is exactly like the **`StackEnumeration`**, with the exception that it supplies a different message string to the **`NullPointerException`**.

To test the **`Queue`** class, we need to modify the declaration in the **`SimpleQueueTest`** again from

```
SimpleQueue sq = new SimpleQueue();
```

to

```
Queue sq = new Queue();
```

The results should be exactly like the original results for the **`VQueue`** and **`SimpleQueue`** tests.

Some Uses for the Queue

A queue can be useful any time data needs to be handled in sequential order. Here are some examples of how we might use a queue:

- **A message queue:** Messages from an outside source can be received and stored in a queue until the application is ready to process them.

- **An event queue:** In GUI environments, input events such as keystrokes and mouse movements need to be handed off to the appropriate handlers in the order in which they are received.
- **A print queue:** In a networked and/or multitasked environment, it sometimes is necessary to share printers between multiple users and print jobs. A print queue, or *spooler*, is used to store print jobs until the resources are available to allow the job to print.

Exercises

1. Add a method to the **`Queue`** class with the following signature:

```
boolean contains( Object o );
```

Have the method non-destructively check the queue to see whether the supplied argument is in the queue. Return **`true`** or **`false`** accordingly.

2. Add a method to the **`Queue`** class with the following signature:

```
int size();
```

Have the method return the number of elements in the queue.

3. Construct an application that implements a message queue. Read messages from a file and display each message in sequence, along with the number of remaining messages in the queue.

Summary

In this chapter, we learned the following:

- Queues are another specialization of the singly-linked list.
- A queue is a *first in, first out* (FIFO) data structure.
- A queue is similar to a stack, because all the operations are performed at the endpoints of the list.
- A queue has only two basic operations: **`put`** and **`get`**.
- The queue is not as efficient as the stack as a vector-based implementation.
- A queue is best implemented as a reference-based list.
- Queues are very common in GUI- and message-based applications.

Chapter 9
Simple Trees

In this chapter, we'll examine the structure and use of simple rooted trees. *Rooted trees* are specialized storage containers that have a single entry point and arrange the elements contained in a hierarchical fashion. We'll draw a comparison between the tree structure and traditional linked lists, such as those covered in previous chapters. We'll take a look at the mechanism behind tree traversal and how it differs from that of the linked list. We'll also briefly look at using an interface to provide generic search-and-compare functionality to the tree.

Trees

For the past several chapters, we've looked at different varieties of linked lists. The linked list was described as a storage mechanism for linear, noncontiguous data collections. In this chapter, we will begin looking at tree structures. The big difference between trees and linked lists is that *trees* store data in a non-linear, non-contiguous fashion. Trees store data nodes hierarchically.

Before we delve into this chapter, we need to know some common terms. A tree node has at most one entry point. An *entry point* is a reference to a node by another node. In Figure 9-1, node A has a reference to node B. This is the entry point to B. A is considered the *parent* of B and the parent of D. B and D therefore are *children* of A. In general, a node can have as many children as desired. Each child can have only one par-

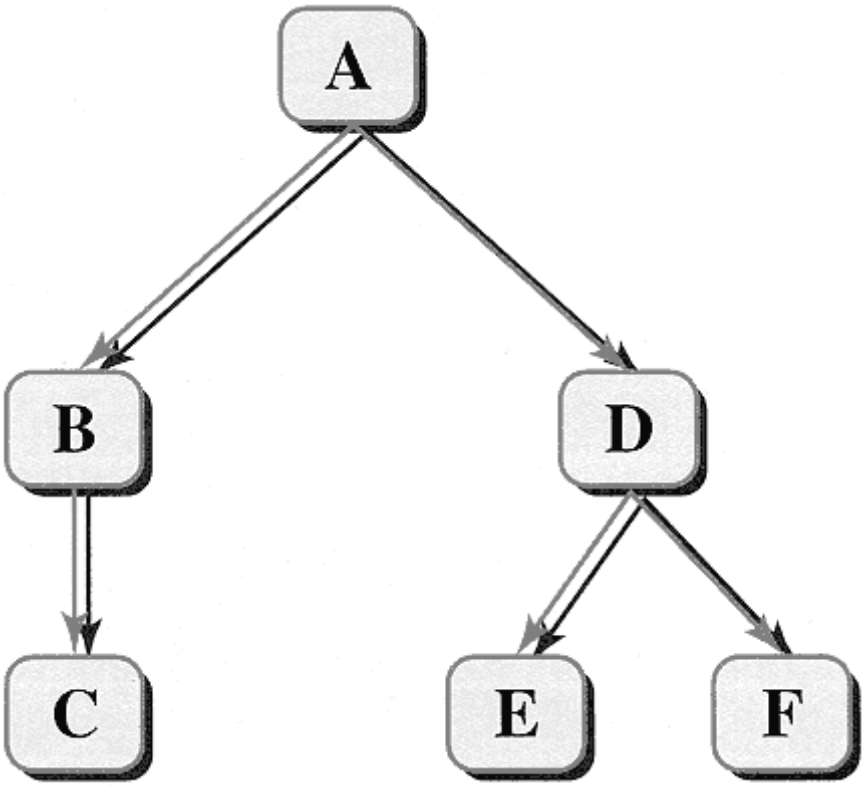


Figure 9-1
A tree with one of its branches filled.

ent, though, and there will always be exactly one less data item or key stored in the node than the number of branches.

A tree also has exactly one root node. The *root node* has no parent and is considered the entry point to the tree. The path between any two adjacent nodes is referred to as a *branch.* In Figure 9-1, node A has two branches. Node B has only one branch. If a node has zero branches (meaning that it has no children), it is called a *leaf* The maximum number of branches a node can have is determined by the type of the tree. Or, more accurately, tree structures are classified by the maximum number of branches each node can support.

A tree has a height. The *height* of a tree is the maximum number of consecutive nodes in the longest path on the tree from root to leaf. In Figure 9-1, the height is three. Along with height, we need to consider level. The *level* is the number of nodes between any node and the root. Nodes B and D are on level 1 of the tree, for example. Nodes C, E, and F are on level 2.

A level is considered to be *full* when it holds the maximum number of nodes it can hold. In the figure, each node can have two possible branches. Level 1 is full (B and D), but level 2 is not. A *balanced* tree is a tree in which each level, with the possible exception of the last, is full. If the tree supported nodes with three possible branches, three nodes would need to be on level 1 in order for it to be full. Six nodes would need to be on level 2 in order to fill that level.

In a tree data structure, the root node is always at the top of the hierarchy. Trees are accessed from the root node at the top, down in the direction of the leaf nodes. Another characteristic of

the tree is that it is a recursive structure. Each node of the tree can be considered the root of its own subtree. Again referring to Figure 9-1, A is the root of the whole tree. D is the root for the subtree containing D, E, and F

Tree Versus Linked List

In the previous chapters that covered the different kinds of linked lists, we saw that the lists were very useful constructs in which to store ordered data. The order was determined by the user manually inserting the data in place. The nature of the tree structure, on the other hand, requires that the order be determined programmatically. The tree's nodes are not stored in a linear fashion; they are inserted into the tree hierarchy based on a comparison to the existing nodes in the tree.

A tree therefore must be supplied with some algorithm by which the data stored in the tree can be compared. The basis on which the comparison is made is arbitrary. The programmer could decide that the data should be compared lexically, numerically, or by some other criteria. There is also no requirement that the data stored in a tree be of a homogenous type. The important factor is that, given any two data items (objects) stored in the tree structure, one can be determined to be greater than, less than, or equal to the other. As long as a consistent basis of comparison is determined, the tree structure's requirements are satisfied.

The basic storage schema in a tree structure uses the comparison relationship between two nodes to determine the position of the nodes in the tree. Suppose that we have a tree node that is defined to have three branches. The left branch of the node contains nodes with data items that compare less than the current node's data. The center branch contains nodes that compare equal to the current node. And the right branch contains nodes that compare greater to the current node.

In a tree containing these types of three branch nodes, the structure of the tree is that nodes on the left side of the tree are comparatively less than nodes on the right. This distinction is arbitrary as well. The nodes just as easily could have been defined so that the right branch holds the less-than nodes and the left branch holds the greater-than nodes. But a distinction does need to be made, and it must be consistent for the entire tree structure. As a general rule, programmers have a tendency to follow the *left is less and right is greater* model. This is probably because people generally sort things from left to right and least to greatest.

Now take a look at the difference in populating a linked list versus a tree structure. Suppose that we have the following data set of city names to be stored:

Chicago

Los Angeles

Atlanta

Boston

Houston

Indianapolis

If we want to add this data to a linked list in alphabetical order, we could do this:

1. Add Chicago to the empty list.

2. Append Los Angeles.

3. Insert Atlanta and Boston before Chicago.

4. Insert Houston and Indianapolis before Los Angeles.

All the determinations of where to insert and add the data can be done outside of the linked list structure.

In a tree, however, the position of the nodes not only depends on the alphabetical order, but also on the order in which they are inserted and the number of branches allowed to each node in the tree. Using the same data set in the same order, inserting the city names into a tree with nodes that have two branches would create a scenario like this:

1. Insert Chicago, which becomes the root node.

2. Insert Los Angeles, which becomes the right node for Chicago.

3. Insert Atlanta, which becomes the left node for Chicago.

4. Insert Boston, which becomes the right node for Atlanta.

5. Insert Houston, which becomes the left node for Los Angeles.

6. Insert Indianapolis, which becomes the right node for Houston.

Figure 9-2 shows the resulting tree structure.

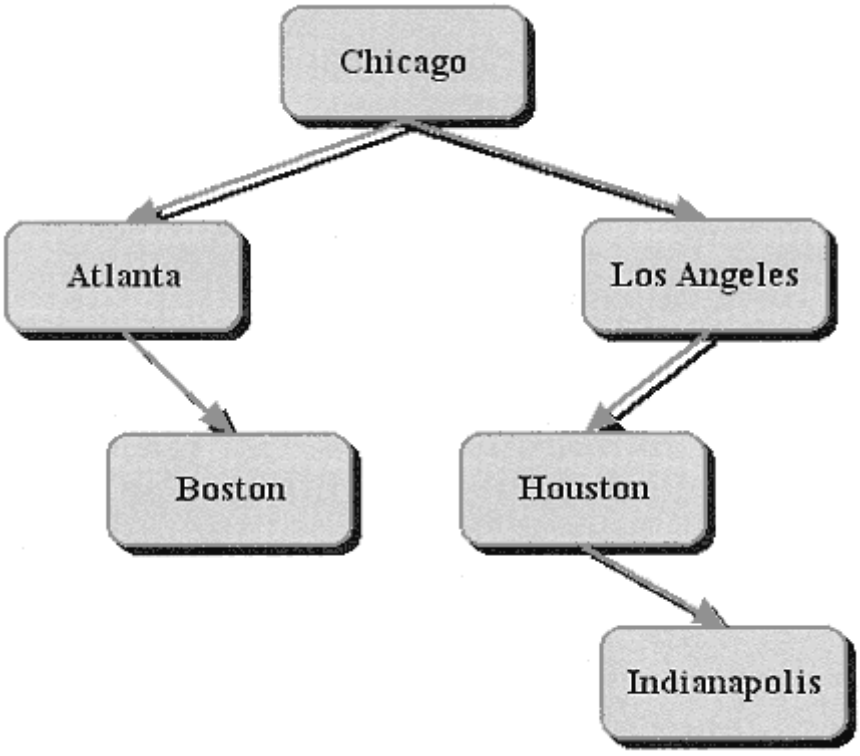


Figure 9-2
The order in which the nodes are added affects the structure of the tree.

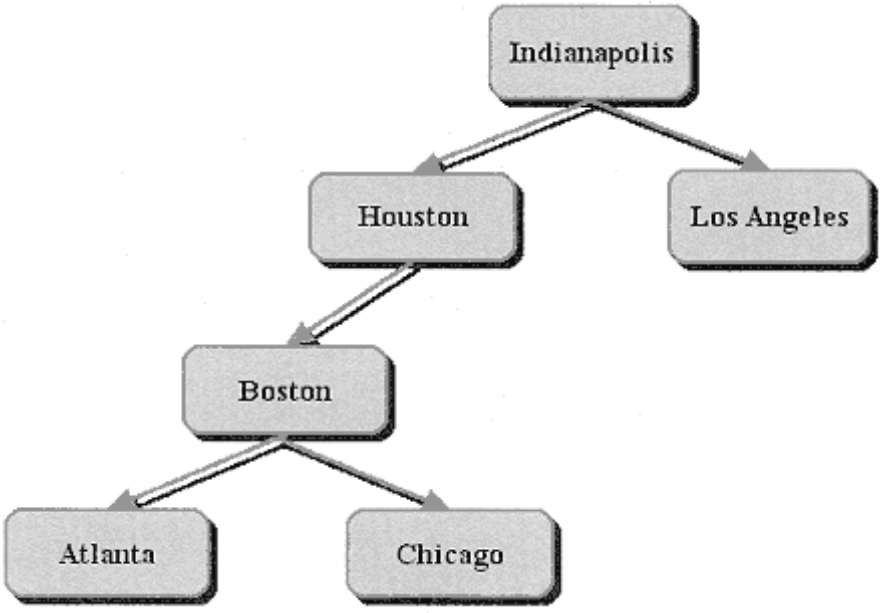


Figure 9-3
The same tree, heavily unbalanced to the left.

Now look at what happens if we insert the nodes into the tree in the reverse order. Figure 9-3 shows the tree generated by the following scenario:

1. Insert Indianapolis, which becomes the root node.

2. Insert Houston, which becomes the left node for Indianapolis.

3. Insert Boston, which becomes the left node for Houston.

4. Insert Atlanta, which becomes the left node for Boston.

5. Insert Los Angeles, which becomes the right node for Indianapolis.

6. Insert Chicago, which becomes the right node for Boston.

The tree in this case looks very different from the tree in Figure 9-2. In fact, though, both trees contain the same data.

Adding Nodes to the Tree

Why do the two sample trees look so different? The answer is in the way in which nodes are added to the tree. As mentioned earlier, a tree has one—and only one—root node. In the case of the first tree, Chicago was the first node added and therefore the root node. In the second tree, Indianapolis become the root node, because it was the first node inserted into the tree.

In the tree in Figure 9-2, the second node added (Los Angeles) is compared to the root node. Because Los Angeles compares alphabetically greater than Chicago, the new node is placed to the right of the root node. Because there are no other nodes on the right branch of the tree, this node becomes the right-hand child of the root node at level 1.

The next node is for Atlanta. Its case is exactly the same as Los Angeles, except that Atlanta is alphabetically less than Chicago. So this node becomes the left-hand child of the root node at level 1. At this point, level 1 is full.

The next node to be inserted in the tree is the one for Boston. Again, it compares less than Chicago, so placement moves to the left branch of the tree. In this case, though, the left child of Chicago already is occupied. The node now needs to be compared to the child, Atlanta. Boston compares as greater than Atlanta, so it is moved to the right branch of the Atlanta subtree. Because this branch is empty, Boston becomes the right child of Atlanta.

Houston is next, and it compares as greater than Chicago. Placement moves to the right branch of the tree where Los Angeles is the child node. The comparison shows Houston to be less than Los Angeles, so Houston is placed as the child on the left branch of the Los Angeles subtree.

The final city is Indianapolis. In the same fashion as Houston, its placement is moved to the right branch of the root tree. Because Houston already occupies the left child position of Los Angeles, the node is compared also to Houston and becomes the child on the right branch of the Houston subtree.

Now we can see why the two sample trees looked so different. Each placement of a new node requires that each node along its path be compared and a decision made as to which branch is next. Also, we should note that the first sample tree is much more balanced than the second. The first tree has a total of two nodes in the left branch and three nodes in the right branch. The

second tree, however, has four nodes in the left subtree and only one node in the right. We'll look at the issue of balance in a moment. But first, we need to look at how we access the nodes in our tree.

Traversal

Much like their linked-list cousins, the nodes of a tree are accessed by traversal. The starting point for the traversal is the root of the tree. Each node in the tree then can be accessed by "walking" the tree structure. In the

linked list, this walking was linear through the list. After a node was visited, it was obvious which node was next in line.

With a tree, there is only one way into each node: from the node's parent. In many cases, though, multiple children exist for a given node and it might not be evident how to access them. The key question here is, "In what order are the nodes accessed?"

The order of the nodes can be determined in several ways. Here, we'll look at the two most common traversal schemes: in-order and pre-order traversal. In all cases, tree traversal begins with the root node. In linked list traversal, after a node is accessed in the traversal, the data in the node is processed immediately, and the traversal moves on to the next node. In tree traversal, a node may need to be accessed several times before the data actually is processed.

There are two operations in a traversal: movement and action. The *movement* operation causes the focus of the traversal to shift to another node. The *action* operation is the processing of the data contained in a node. The movement operation can happen multiple times for a single node in a single traversal. This is not as confusing as it sounds. First, take a look at how tree traversal works.

In-Order Traversal

Using in-order traversal processes the nodes in exact sort order; this is true regardless of the number of child nodes. The key concept to remember is that, from the perspective of the nodes, *left* means *less than* and *right* means *greater than.*

To access the nodes in sort order, we need to start with the "least" or leftmost node and work our way through to the "greatest" or rightmost node. This is the same general principle we might find in a linear list that is accessed from left to right, from the least to greatest value node, according to the comparison criteria. In a tree, however, we are working in more than one dimension. Besides left to right, there is also top to bottom.

In the case of the tree, the top is the root node, and the leaves are at the bottom. To find the first (least) node, we start at the root node. If there is a populated left branch to the root node, there are nodes that evaluate less than the root. Take another look at the first sample tree, which is shown again in Figure 9-4, to understand this process.

Start with the root node, Chicago. There is a left branch to the Chicago node, so we move there. Now the current node is Atlanta. There is no left branch, so Atlanta must be the leastmost node. At this point, we take ac-

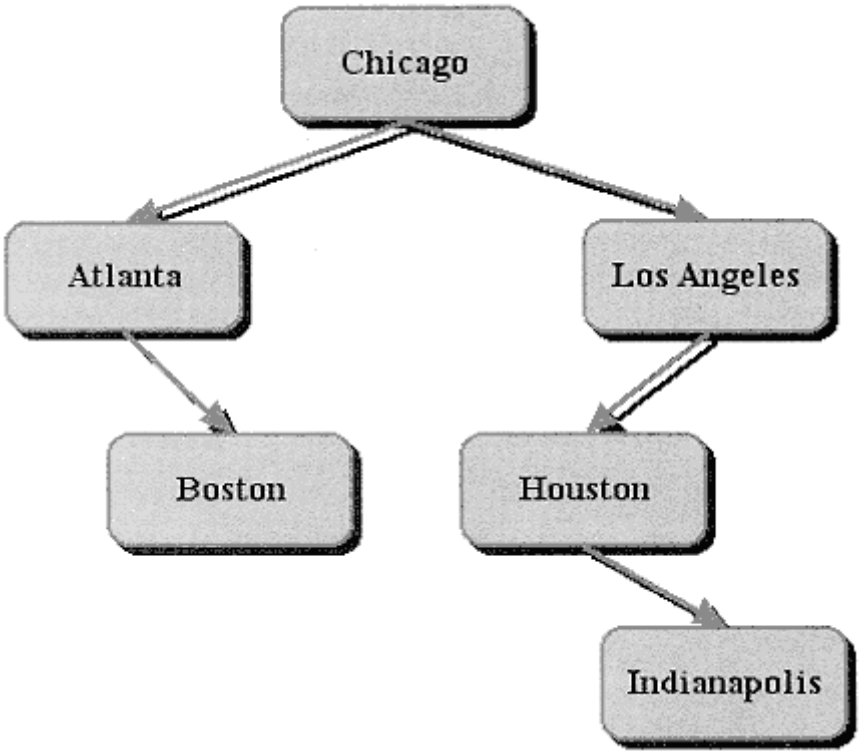


Figure 9-4
Each subtree is traversed in total before its parent.

tion and process the node's data. Suppose that we are printing the list. In this section, each city name appears in the order in which it will be printed. After each city name, we'll see an explanation of why that name is printed in that order.

```
Atlanta
```

If any nodes are greater than Atlanta but still less than Chicago, they will reside in the right branch of the Atlanta node. We next move to Boston. If there are nodes less than Boston but greater than Atlanta, they would reside in the left branch of the Boston node. Because the left branch is empty, we process the data for the current node.

```
Boston
```

If the right branch of the Boston node were populated, it would be processed next. Because the branch is empty, we move back up the tree to the root node. Now that the root node's entire left branch has been processed, it's data can be processed as well.


```
Chicago
Houston
```

At this point, it is time to process the right branch of the tree. The right child of Chicago is the Los Angeles node. Los Angeles has a left child, Houston, which must be processed first. So

Houston is made the current node. Because Houston does not have a left child, Houston's data is processed.

```
Indianapolis
```

The Houston node has a right branch. The Indianapolis node is processed next. It has no children, so it finishes the processing of the left branch of the Los Angeles node.

```
Los Angeles
```

Because Los Angeles does not have a right branch, it is the last (greatest) node processed.

Pre-Order Traversal

Pre-order traversal works in much the same fashion as in-order traversal. The only real difference between the two is that, in pre-order traversal, the data in the node is acted on before either of the branches is examined.

In other words, the in-order scheme follows:

1. Process the left branch.

2. Process the current node.

3. Process the right branch.

The pre-order scheme follows:

1. Process the current node.

2. Process the left branch.

3. Process the right branch.

Although the movement between the nodes of the tree remains unchanged, the order in which the data is processed varies. The output for a pre-order traversal is far different from the previous in-order example, as shown in Table 9.1.

As we can see, the order in which the nodes are processed has a great impact on the behavior of the traversal operation. The thing to keep in

TABLE 9.1

In-Order Traversal	Pre-Order Traversal
Atlanta	Chicago
Boston	Atlanta
Chicago	Boston
Houston	Los Angeles
Indianapolis	Houston
Los Angeles	Indianapolis

mind here is that, for the most part, the users of a tree are interested only in the ordered traversal. The pre-order traversal generally is used for internal operations, such as searching the tree for a particular node. It is beneficial in many operations, such as searching, to examine each node as we pass though it. This can increase the performance of the search operation.

Rotation

The next tree concept we need to explore is rotation. As we saw with the first two tree examples, the order in which the data is added to a tree can have a great impact on the actual structure of the tree. It is quite possible to generate a tree in which one branch is much more populated than the other. This causes the tree to become unbalanced, which affects the performance of the add and search operations.

In a worst-case scenario, where the data inserted into a tree already is in order, the resultant tree ends up being a linear construct similar to the linked list. In the example in Figure 9-5, the nodes were added in alphabetical order: A B C D E F. Each node was placed as the right child of the node before it.

The resulting tree ends up being six levels high. It would be considerably better to have this tree fill the lower levels instead of continuously adding higher and higher levels. If the levels of this tree were filled, the tree would end up three levels high instead of the six shown. Refer back to Figure 9-1 for an example of what a properly balanced tree of this size should look like.

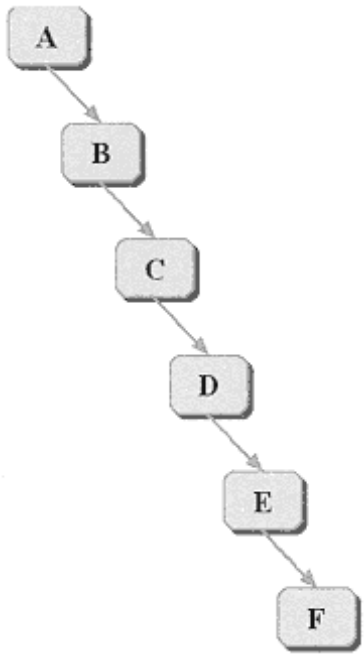


Figure 9-5
A worst-case tree.

To further demonstrate the problem with this linear example, consider what would happen if the rest of the letters of the alphabet were added as nodes, in order. Our tree would end up

being 26 levels high; the equivalent balanced tree with two branches per node would be only five levels high.

To correct an unbalanced tree, we need to modify the structure of the tree without destroying the order of the nodes. This is where rotation comes in. Because the structure of the tree is based on well-defined properties, it is possible to shift the nodes and retain their original order.

Here, we will take a specific subtree and rotate it so that the level it represents is more balanced. Take a look at the examples shown in Figures 9-6 and 9-7.

Figure 9-6 shows how an unbalanced right branch in a two-branch node can be rotated to the left to achieve balance. The C node, which originally was the root of the subtree, has been moved to the left child of the D node. The D node now becomes the new root of the subtree. The net effect of this rotation of position is that it reduces the number of levels in the subtree and fills the reduced level. The process is largely the same for nodes with more than two branches. The nodes can be shifted in position, but the greater than/less than relationship must be maintained at all times.

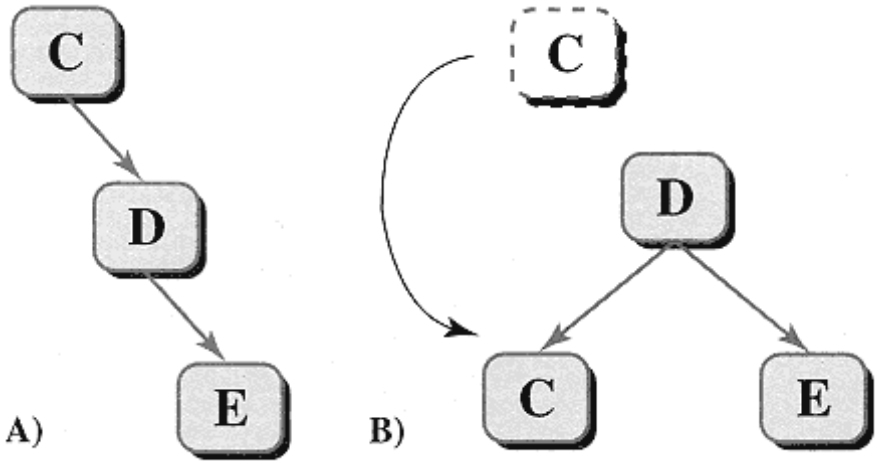


Figure 9-6
A left rotation.

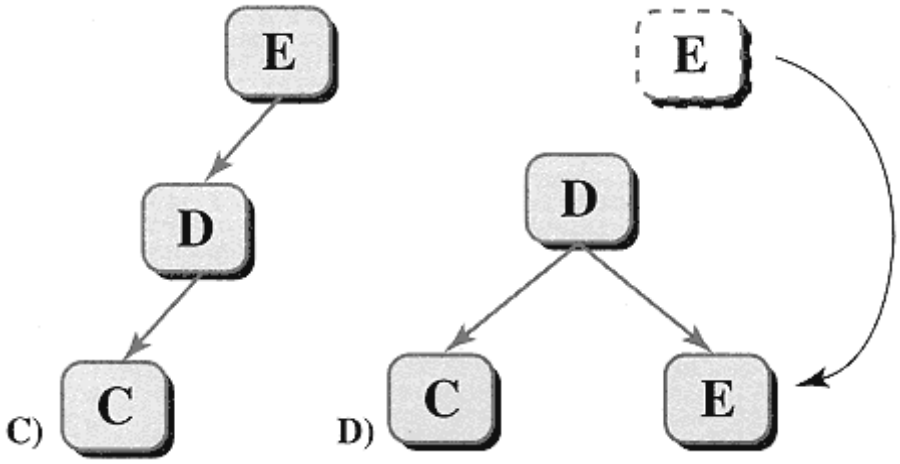


Figure 9-7
A right rotation.

Figure 9-7 demonstrates a rotation to the right. The original subtree is unbalanced to the left. The E node, which is the root of the subtree, becomes the right child of the D node. The D node becomes the new root of the subtree. In both this case and in the left-rotation example, we can say that we rotate around the D node, or that D is the *axis of rotation.*

There is one important restriction on this type of rotation. The node being used as the axis of the rotation cannot already have a child in the direction of the rotation. In the left-rotation example, if the D node already had a child on its left branch, the rotation could not be performed in this manner. It still is possible to perform such a rotation, but it becomes more complex to do so.

Now look at the effects of rotation on an unbalanced tree. To balance the tree shown in Figure 9-5, for example, we will want to rotate the tree to the left, because all the nodes are on the right branch of the tree. To balance the tree and reduce its six levels, we'll perform left rotations on the root of the tree until the branches of the tree are as even as possible. Our axis in these rotations always will be the right child of the tree's root node.

Our first rotation is a left rotation using the B node as the axis. The result of this rotation is that the B node becomes the new root of the tree, and the A node is the left child of the B node. We still have four nodes on the right branch of the tree and only one on the left (not counting the root node, which is at the center).

Because we have a total of six nodes, and six nodes can fit on two levels, we can go ahead and perform another left rotation to fill the second level. This time, we'll use the C node as the axis. The C node resides in the same place that the B node did in the previous rotation—the right child of the root node of the tree. Figure 9-8 shows the effect of this rotation.

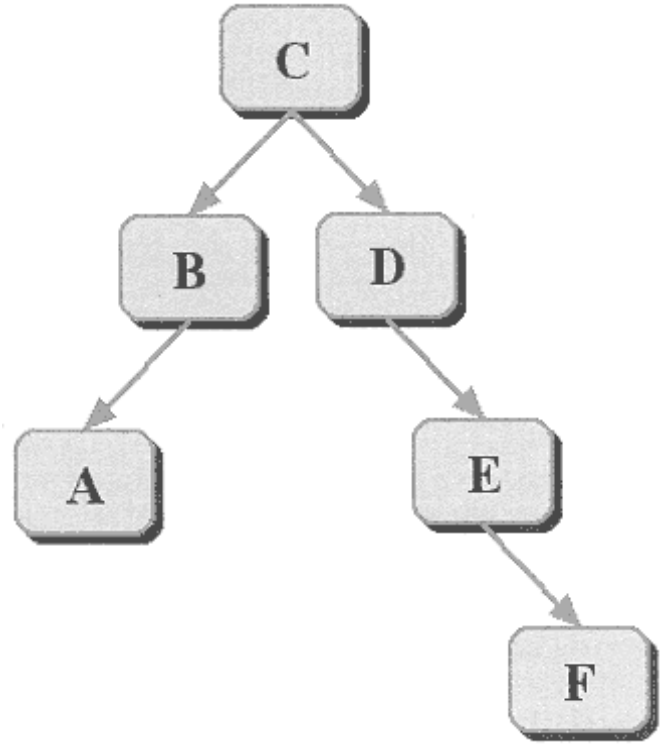


Figure 9-8
The tree after the second left rotation.

The C node is the new root of the tree. The B node has been moved to the left child of the C node. Notice that the A node continues to be the left child of the B node. Nothing about the rotation has changed the relationship between A and B. At this point, we have two levels on the left branch of the tree and three levels on the right branch.

Performing another rotation from the right child of the root will only switch the imbalance from the right to the left branch. We still have too many levels on the right branch, though. There is a third level, even though the second level is not filled. To remedy this, we will move the rotation on the next level of the branch. We will perform another right rotation using the E node as the axis. This rotation affects the subtree rooted at the D node.

This rotation causes the D node to become the left child of the E node. The E node replaces the D node as the root of the subtree, which yields the well-balanced tree shown in Figure 9-9.

The three rotations we performed transformed the heavily unbalanced tree we began with into the completely balanced (but not full) tree in Figure 9-9. This kind of internal balancing is transparent to the user and has no effect on in-order traversals. There would be a definite change in the results of a pre-order traversal but, as mentioned earlier, pre-order traversals generally are used only internally, anyway.

Now that we've examined the basic concepts of the tree structure, we will move on to the next few chapters and look at different kinds of trees and their implementations.

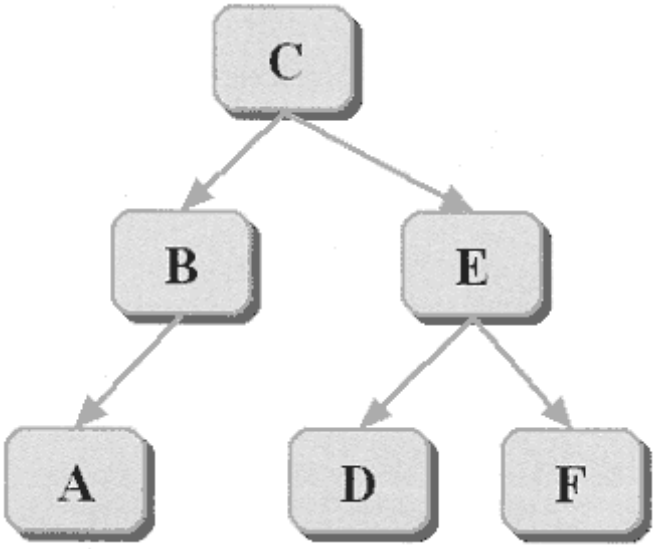


Figure 9-9
The final structure of the balanced tree.

Page 168

Exercises

1. Diagram three trees using nodes that have two branches each. Use the following data sets:

{ A, B, L, W, Q, S, Z, P, I, R, E, X, C, J}

{ Z, Y, X, W, V, U, T, S, R, Q**}**

{ A, B, C, D, H, G, F, E, I, J, K, L, P, O, N, M}

2. Diagram the rotations necessary to balance each of the trees we created in Exercise 1.

Page 169

Summary

In this chapter, we learned the following:

- Trees are similar to linked lists, because they use data nodes to abstract the tree structure from the tree implementation.
- Unlike linked lists, trees are not linear, and the mechanism for data placement is integral to the tree itself.
- Trees are recursive structures, because each node of a tree is the root of its own subtree.
- The actual structure of a tree can vary based on the order in which data nodes are added to the tree.
- Similar to linked lists, the data in a tree generally is accessed by the user traversing the tree

and performing some action on each node in the tree.

- The standard way for a user to traverse the tree is called *in-order* traversing, which processes nodes in sorted order.
- It sometimes is necessary to manipulate the tree structure to achieve balance.

Chapter 10
Binary Trees

This chapter expands on the concepts presented in Chapter 9 and explains the binary tree. A *binary tree* is implemented with a balanced tree structure to improve performance. This chapter also explains implementing the search algorithm and contrasts it with using the sequential search available in non-sorted linked lists.

Binary Trees

Chapter 9 explained the base functionality of the basic tree structure. To keep the explanation simple, the examples were based on tree nodes with a maximum of two branches. This type of tree is called a *binary tree* and will be the focus of this chapter Although a tree structure may support nodes with any number of branches, the binary tree is the simplest to implement and in many ways the most efficient in performing its operations. The operations are simplified by restricting each decision in the tree to a two-way choice: left/right or greater/less.

Tree Nodes

Chapter 9 explained the concepts involved with the tree. This chapter will concentrate on the implementation of the binary tree. The first step in implementing the tree in Java is to create the node class (see Figure 10-1). The binary tree node class is very similar to the doubly-linked list node class used in the **`DLinkedList`** examples.

Figure 10-1
TreeNode.java.

```
package adt.Chapter10;

public class TreeNode
{
    public TreeNode( Object o )
    {
        left = right = null;
        data = o;
    }

    public Object getData()
```

```
    {
        return data;
    }

    public TreeNode getLeft()
    {
        return left;
    }

    public TreeNode getRight()
```

Continues

Figure 10-1
TreeNode.java.
(continued)

```
    {
        return right;
    }

    public void setData( Object o )
    {
        data = o;
    }

    public void setLeft( TreeNode l )
    {
        left = l;
    }

    public void setRight( TreeNode r )
    {
        right = r;
    }

    public String toString()
    {
        return "TreeNode " + data;
    }

    TreeNode left;
    TreeNode right;
    Object data;
}
```

The **`TreeNode`** class follows the format of the linked list nodes. The main difference is in the names of the link references: right and left. Of course, there is no functional difference in the references. The name change just makes the code easier to read. As is usual, we use proper data encapsulation by making the member fields private and supplying accessor methods to manipulate the values.

An Interface to Compare Nodes

Now that we have a safe, generic node class, we need to address some of the other issues involved in the tree implementation. For the tree to correctly place the **`TreeNode`**s in sort order within the tree, we need to be able to compare two nodes quantitatively (see Figure 10-2).

Figure 10-2
Comparable.java.

```
package adt.Chapter10;

public interface Comparable
{
    public int compare( Object a, Object b );
}
```

Figure 10-3
Traversal java.

```
package adt.Chapter10;

public interface Traversal
{
    public void process(Object o);
}
```

To compare the nodes, we'll create an interface that defines a method the tree class can call when it needs to compare two objects. It will be the responsibility of this method to be able to determine whether a node is greater than, less than, or equal to another node. In the examples, all the data is in the form of alphanumeric strings. In such cases, the **`compare()`** method could just make a call to **`String.compareTo()`** to perform the comparison.

A Tree Traversal Interface

We are going to need one more interface defined for the tree (see Figure 10-3). Remember that, during traversal, there is a movement operation and an action operation. The action operation needs to be defined externally just as the **`compare()`** method was.

The **`Traversal`** interface defines a method called **`process()`**. The instantiator of the tree will need to pass a class that implements the **`Traversal`** interface to the tree object. The tree will call this method when it is time to process the data in the node.

The Tree Class

Now we can get on to the **`Tree`** class itself. Figure 10-4 shows the entire source listing for the **`Tree`** class. The tree has two private instance variables:

Figure 10-4
Tree.java.

```
package adt.Chapter10;

public class Tree
{
    public Tree( Comparable o )
    {
        c = o;
    }

    public void add( Object o )
    {
        add( root, new TreeNode( o ) );
    }

    protected void add( TreeNode root, TreeNode newNode )
    {
        if( root == null )
        {
            this.root - newNode;
            return;
        }

        int val = c.compare( newNode.getData(),
             root.getData() );

        if( val == 0 )
        {
            root.setData( newNode.getData() );
            return;
        }
        else if( val < 0 )
        {
            if( root.getLeft() == null )
                root.setLeft( newNode );
            else
                add( root.getLeft(), newNode );
        }
        else if( val > 0 )
        {
            if( root.getRight() == null )
                root.setRight( newNode );
            else
                add( root.getRight(), newNode );
        }
    }

    public Object search( Object o )
    {
        return search( root, o );
    }
```

Continues

Figure 10-4

Continued

```
    protected Object search( TreeNode root, Object o )
    {
        if( root == null )
        {
            return null;
        }

        int val = c.compare( o, root.getData() );

        if( val == 0 )
        {
            return root.getData();
        }
        else if( val < 0 )
        {
            return search( root.getLeft(), o );
        }
        else if( val > 0 )
        {
            return search( root.getRight(), o );
        }
        return null;
    }

    public void traverse( Traversal t )
    {
        traverse( INORDER, t );
    }

    public void traverse( int type, Traversal t )
    {
        traverse( root, type, t );
    }

    protected void traverse( TreeNode root, int type,
         Traversal t )
    {
        TreeNode tmp;

        if( type == PREORDER )
            t.process( root.getData() );

        if( (tmp = root.getLeft()) != null )
            traverse( tmp, type, t );

        if( type == INORDER )
            t.process( root.getData() );

        if( (tmp = root.getRight()) != null )
            traverse( tmp, type, t );
    }

    protected TreeNode root;
```

Continues

Figure 10-4
Continued

```
    protected Comparable c;

    public final static int INORDER = 1;
    public final static int PREORDER = 2;

    protected final static int RIGHT = 1;
    protected final static int LEFT = 2;
}
```

root and **c**. The **root** variable is used to hold a reference to the root of the tree. The **c** variable holds a reference to the object that implements the **Comparable** interface that will be used in all the placement and movement operations.

Only one constructor is supplied with the **Tree** class. The constructor takes a **Comparable** object as an argument, which is used to initialize the **c** reference. By not supplying a default constructor, public **Tree(),** the **Tree** class forces all new trees to be supplied with the **Comparable** object.

Besides the constructor, the remaining methods are **add(), search()** and **traverse().** The interesting thing in this class is not only that these three methods are overloaded, but also that some of the overloaded methods are protected and therefore not available to other classes (unless the class is a subclass or a part of the same package). Now take a look at why the class was designed this way.

As a public user of the **Tree** class, an application most certainly will need to add data (nodes) to the tree. Eventually, the client application will need to search for data in the tree, traverse the tree, or both. In the case of the add operation, there is one public **add()** method. The only required argument for the public **add()** method is the object that is to be stored. But internally, the class uses recursion to properly place the node within the tree.

To use recursion in the **add()** method, the class requires a relative root node for the tree or subtree. The external user of the tree doesn't need to know anything about nodes, roots, or subtrees to use the tree. In fact, it is better encapsulation to insulate the user from these details. Therefore, the protected **add()** method tracks the root of the tree and is called from the public **add()** method.

Adding Nodes to the Tree

The protected **add()** method takes a **TreeNode root** and a **TreeNode newNode** as its arguments. The **newNode** is the data object passed in from

the public **add()** method embedded in a new **TreeNode** object. This node needs to be passed along and placed in the proper position in the tree. The **root** argument, however, is the key to the recursion. When the protected **add()** method is first called from the public **add(),**

this **root** is the actual root of the tree. If the tree is empty, this root will be **null**, and we know to make this node to the root node. The new **TreeNode** is assigned to **this.root**. The **this** must be used explicitly in this case, because the **root** argument conflicts with the class's **root** instance variable. After this is done, the add operation is complete, and the method returns.

If the tree already is populated, a little more work must be done. We need to figure out exactly where in the tree structure this data node belongs. To do this, we need to compare the data in the node provided by the caller to the data object in the root node. There are three possibilities: The data could compare greater than, less than, or equal to the root's data object: Depending on the outcome of the comparison, the new node is placed in the left or right branch from the root node.

The comparison is performed via the **compare()** method in the **Comparable** object supplied in the **Tree** constructor. The **compare()** method is defined to return an int value that represents the relationship between the two objects. A negative value indicates that the new object compares as less than the root's data object. Zero means that the objects are equal, and a positive value means that the new object is greater than the root's object.

If the objects compare as equal, there is a problem. The tree is structured so that it cannot handle duplicates. Theoretically we could treat a duplicate by default as less than or greater than and just insert it into the tree at the appropriate location. The new node would be inaccessible to the search operation, though, because the first instance of the object (the root's data object, in this case) would be a match. To avoid this problem and all the associated issues that would come up, we have two choices. We can throw an exception or treat this operation as an update. In this case, the update option was chosen, and the data in the **root** node is assigned the data from the new node. This might be slightly confusing to think about, but remember that the tree has no control over the **compare()** method. The data object in the node may be a complex object, and only the keys of the objects are compared, such as in an address record in which only the names are compared.

If the new node's data object compares as less than the root's data object, the new node needs to be placed somewhere on the left branch of the tree. If there is no left child of the root node, the new **TreeNode** becomes the left child. If there is already a left child, we need to determine

where on the left branch the new node goes under the child. This is where the recursion comes in.

In Chapter 9, we learned that trees are *recursive* structures, which means that each node in the tree is a subtree and can be considered to be a complete tree in and of itself. The structural recursion of trees is crucial to the workings of the tree methods; because a subtree is also a tree, any operation that can be performed on a tree can be performed on a subtree. To illustrate, look again at the protected **add()** method in Figure 10-4. The **TreeNode root** argument just as easily could be the root of a subtree as the root of the entire tree. We take advantage of this fact to recurse down the left branch to place the new data node.

The left child of the root node is used as the *root* argument in a new call to the protected

add() method. The entire process begins again with the call to the **compare()** method. The recursive call will never have a **null root,** because it already has been established that a node is being passed. This time, the comparison is between the new data node and the root of the subtree. The root node of the subtree, in this case, is the left child of the entire tree's root.

This recursion continues until the proper place for the data node is determined and a new **TreeNode** is added to the tree. Each recursion through the protected **add()** method is for a subtree of the next level of the tree. We can follow the path through the tree by tracing through the illustration in Figure 10-5.

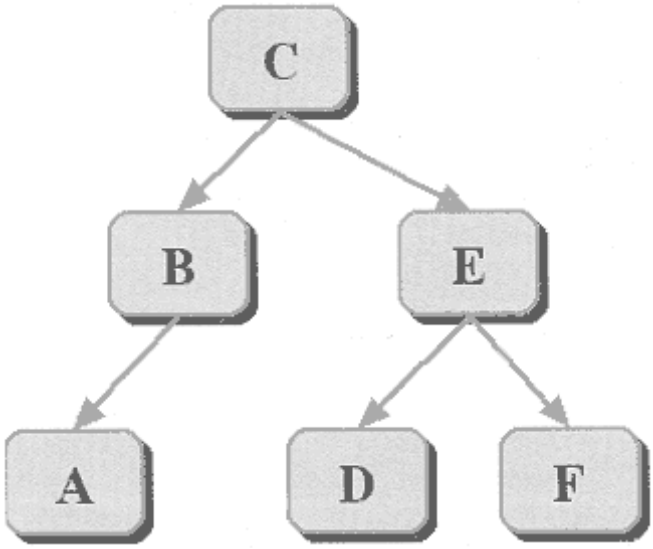


Figure 10-5
Each node can be considered to be the root of its own subtree.

Of course, the entire scenario is the same for a positive return value from **compare().** The comparison is greater than instead of less than, and the child is on the right branch instead of the left. But the processing is the same on either branch, and at every recursion the placement may move left or right.

Searching the Tree

The next overloaded method of the **Tree** class we'll look at is the **search()** method. In comparison to the linear search used with linked lists, a binary tree search is extremely fast and efficient. In the worst-case scenario, the number of lookups required to find a node is equal to the number of levels in the longest branch in the tree. In the linked list, though, the worst case requires as many lookups as there are nodes in the list.

As with the **add()** method, there is a public and a protected **search()** method. The public **search()** method takes only an object as an argument. This is the object to compare to the data objects in the nodes. When a match is found, the data object for that node is expected to be returned. If no match is found in the tree, the method returns **null.**

The public **search()** method, like the public **add()** method, has only a single line of code—a call to the protected **search()** method. In this respect, the **search()** methods

work the same way as the **`add()`** method. The user doesn't need to know about **`TreeNode`**s to find the node for which the user is searching. For the search to be recursive, though, the protected method needs to have a root node from which to search.

In most aspects, the protected **`search()`** method is just like the protected **`add()`** method. The big difference is that, instead of adding the object to the end of the branch, the search method walks the branch only as far as necessary to identify and return the data object in question.

Traversing the Tree

The last of our overloaded methods is the **`traverse()`** method, for which there are two public methods and one protected method. The first public **`traverse()`** method takes an object that implements the **`Traversal`** interface as its only argument. The **`Traversal`** object provides the **`process()`** method, which performs the user's desired action on the data

object in each node in the tree, in order. This method assumes that the user wants to use in-order traversal.

The second public **`traverse()`** method takes a **`Traversal`** argument again, as well as an int argument: **`type`**. The **`type`** argument lets the user specify the type of traversal to use. This gives the user access to the preorder traversal and makes it easier in the future to expand the set of traversals offered. Both this and the first public traversal method make a single call to the protected **`traverse()`** method.

The protected **`traverse()`** method takes three arguments: a **`TreeNode root`**, an int **`type`**, and a **`Traversal t`**. As with the **`add()`** and **`search()`** methods, the protected **`traverse()`** method requires the **`root`** argument in order to recurse. The **`type`** and **`t`** arguments are passed through from the public methods. The traversal operation begins with the determination of the type of the traversal. If the type is pre-order, the node is processed first. If the type is in-order, we first move up the left branch. At this point, we recurse up the left branch. The **`traverse()`** method is called again, with the left child as the root node.

Assuming that the traversal type is in-order, the recursion continues up the left branch until there is no left child. We are at the leastmost node at this point. The node is processed by calling the **`process()`** method from the **`Traversal`** object **`t`**. If there is a right child to the current node, the recursion moves up that branch in the same manner.

Traversal is different from the add and search operations, because not only do we need to travel up the branches, but we also need to go back down the tree to process every node. After we've exhausted the upward recursive movement, we need to go back. The **`TreeNode`**s do not have references to their parents, so how do we go back? If we look at how we got to the current node, the answer becomes clear. With each movement, the **`traverse()`** method was called from the **`traverse()`** method. When the method returns, it returns to the calling method, which has a root one level higher. In this fashion, we can move up and down the branches to visit each node.

Using the Tree

Now that we've examined the basic operations, it's time to put the tree to work. We'll use the cities example we've been looking at so far. Here, we're going to create an application to populate and traverse the tree. Figure 10-6 shows the **`TreeTest`** class source listing.

Figure 10-6
TreeTest.java.

```
package adt.Chapter10;

public class TreeTest
{
      public static void main( String args[] )
      {
         Tree t = new Tree(
             new Comparable()
             {
                 public int compare( Object a, Object b )
                 {
                     return ((String)a).compareTo(
                         (String)b );
                 }
             }
         );

         t.add("Chicago");
         t.add("Los Angeles");
         t.add("Atlanta");
         t.add("Boston");
         t.add("Houston");
         t.add("Indianapolis");
         t.traverse(
             new Traversal()

             {
                 public void process(Object o)
                 {
                     System.out.println( o );
                 }
             }
         );

         System.out.println( "SEARCH=" + t.search(
              "Houston" ) );
         System.out.println( "SEARCH=" + t.search( "Miami"
              ) );

      }
}
```

The **`TreeTest`** class has only one method: the **`main()`** method. When the application starts, it first creates a **`Tree`** object. The **`Tree`** constructor requires a **`Comparable`** object as an argument. In this case, we supply an anonymous inner class that implements the **`Comparable`** interface. Inner classes were introduced to the Java language in Version 1.1. An anonymous

inner class makes it easy to implement an interface that doesn't have a lot of methods defined and doesn't need to be referenced elsewhere in the defining class.

Because the data in the cities example consists entirely of **`String`** objects, the **`compare()`** method defined just calls the **`String.compareTo()`** method to provide the comparison.

After the **`Tree`** object is created, the application moves on to add all the city data. As we can see, the user of the class can remain blissfully unaware of the structure of the tree itself. There is no external knowledge of the placement of the nodes.

After all the data is added, **`TreeTest`** performs a traversal of the tree. The **`Traversal`** object supplied as an argument to the public **`traverse()`** method is another anonymous inner class. Again in this case, the anonymous inner class is appropriate, because the **`Traversal`** interface defines only one method, and the **`TreeTest`** class doesn't need to refer directly to the **`Traversal`** object anywhere. The **`process()`** method simply outputs the name of the city to the console.

Finally to test out the **`search()`** method, we look up two cities. Houston is one of our test cities and so is found. Miami doesn't exist in the tree, so the **`search()`** method returns **`null`**.

Balancing the Tree

One of the problems with our binary tree implementation is that the physical structure of the tree is dependent on the order in which the data is added to the tree. In the worst-case scenario, the data could be added to the tree already in sorted order. This would cause the tree to resemble a singly-linked list with all the additions extending the right branch only. In this case, all the advantages of the binary tree over the linked list are lost.

For the tree to be efficient, it needs to be reasonably balanced. It is desirable to have as few levels as possible for the data set being stored and to have those levels as full as possible. It is highly unlikely that the data will be added to the binary tree in precisely the right order to achieve perfect balance to the tree. It therefore is possible to monitor and adjust the tree's balance at the time that data is added to the structure to achieve a better balance. We'll investigate other tree structures later that can lead to trees that are more nearly perfectly balanced. At this point, we are going to keep it simple, though, and just try for a tree that isn't too lopsided.

As discussed in the preceding chapter, balance is achieved through the use of the rotation operation. Any binary tree or subtree can be rotated to the right or the left around any node, as long as the branch opposite the direction of rotation is not empty. Basically, this means that because

the left child becomes the new root of the subtree in a right rotation, we can't perform the rotation if that child is empty. The same goes for a left rotation and right child.

Now take a look at the **`Tree`** class with some measure of automatic balancing built in. To

build the **`BalTree`** class, as shown in Figure 10-7, we will extend the **`Tree`** class we just created. For the most part, we simply are going to reuse the methods defined in the original **`Tree`** class.

Figure 10-7
BalTree.java.

```
package adt.Chapter10;

public class BalTree
     extends Tree
{
     public BalTree( Comparable o )
     {
         super( o );
     }

     public void add( Object o )
     {
         super.add( o );
         if( root != null )
         {
             root = balance( root );
         }
     }

     protected int branchCount( TreeNode root, int
          direction )
     {
         count = 0;
         TreeNode branch = null;

         if( root == null )
             return 0;

         switch( direction )
         {
             case RIGHT:
                 branch = root.getRight();
                 break;
             case LEFT:
                 branch = root.getLeft();
                 break;
         }

         if( branch == null )
             return 0;
```

Continues

Figure 10-7
Continued.

```
        traverse( branch, INORDER,
```

```
            new Traversal()
            {
                public void process( Object o )
                {
                    count++;
                }
            }
        );

        return count;
    }

    protected TreeNode rotate( TreeNode root, int
         direction )
    {
        TreeNode newRoot = null;
        TreeNode orphan = null;

        switch( direction )
        {
            case RIGHT:
                newRoot = root.getLeft();
                root.setLeft( null );
                orphan = newRoot.getRight();
                newRoot.setRight( root );
                break;
            case LEFT:
                newRoot = root.getRight();
                root.setRight( null );
                orphan = newRoot.getLeft();
                newRoot.setLeft( root );
                break;
        }

        if( newRoot == null )
            return root;

        if( orphan != null )
            add( root, orphan );

        return newRoot;
    }

    protected TreeNode balance( TreeNode root )
    {
        if( root == null )
            return null;

        int left = branchCount( root, LEFT );
        int right = branchCount( root, RIGHT );
```

Continues

Figure 10-7
Continued.

```
        if( left > right )
        {
            while( left > right + 1)
            {
                root = rotate( root, RIGHT );
                left = branchCount( root, LEFT );
                right = branchCount( root, RIGHT );
            }
        }

        if( right > left )
        {
            while( right > left + 1 )
            {
                root = rotate( root, LEFT );
                left = branchCount( root, LEFT );
                right = branchCount( root, RIGHT );
            }
        }

        root.setLeft( balance( root.getLeft() ) );
        root.setRight( balance( root.getRight() ) );

        return root;
    }

    protected int count = 0;
}
```

The constructor for the **`BalTree`** class takes a **`Comparable`** object as an argument, just like the superclass **`Tree`.** All this constructor needs to do is call the superclass constructor and initialize the new instance variable, **`count`.** The **`count`** field will be used to help judge whether the tree is out of balance.

The public **`add()`** method of the superclass is overridden in the **`BalTree`** class. At this point, we attempt to balance the tree. By balancing the tree after the add operation, we accomplish two things. First, we ensure that the tree always will be reasonably in balance. The only way that data is added to the tree is through this method, and adding data is what throws the tree out of balance.

The second thing that balancing from the add method gives us is a centralized point of operation. This way, we don't have to worry about calling the balance routine before each access by the user. This is somewhat a matter of judgment, however. Shifting the balance operation to the point of user access (the search and traverse operations) may have some advantages as well. If the data stored in the tree were added in a relatively

random order, it tends to be more efficient to balance the tree after the tree is fully populated.

A **`count`** method is used internally to discover how far out of balance the tree is. A balanced binary tree will have roughly the same number of nodes in each of its two branches. This method leverages the **`traverse()`** method to do its counting. Remember that each node can

be considered to be a complete subtree in and of itself By completely traversing one of these subtrees and keeping a count as we go, we can determine exactly how many nodes are in the branch in question. This doesn't include the parent node, of course.

The **branchCount()** method takes two arguments: a **TreeNode root** to be used as the parent for the count and an int constant representing the *direction* of the branch to count **(LEFT** or **RIGHT**). The appropriate child of the parent is used as the root for an in-order traversal. The **Traversal** process method is used to increment a counter to keep track of the number of nodes visited during the traversal; this yields the correct count for the branch. The count does not include the node passed to the method as the parent.

The **rotate()** method is used to manipulate the structure of the tree to bring it more in balance. It is important that the rotation operation does not impact the results of an in-order traversal of the tree. The **rotate()** method takes two arguments: a **TreeNode root** and an int constant to indicate the *direction* of the rotation. The **root** node is the node that is actually going to be rotated into a new position. The rotations for left and right are mirrors of each other. For convenience, this section covers only the left rotation, but keep in mind that the steps for a right rotation are equivalent and opposite (refer to Figure 10-7).

We should be concerned with three nodes in a left rotation. The **root** passed to the method is the node that is targeted to be moved. The **TreeNode newRoot** is the right child of **root.** It will take **root**'s place in the tree structure when the rotation is complete. The **TreeNode orphan** is the node, if any, that will be displaced as **newRoot**'s left child by **root** moving into its place.

The actual Java code for the rotation is fairly straightforward. First, **newRoot** is assigned the right child of **root.** The **right** reference in **root** then is cleared so that there isn't an accidental circular reference in the tree. Next, **orphan** is assigned to the left child of **newRoot.** The rotation is going to cause **root** to become the left child of **newRoot,** and we don't want to lose the subtree rooted at **orphan**. We then go ahead and set **root** as the left child of **newRoot**. If **orphan** is not **null**, a node and possibly an entire subtree needs to be put back in the tree. Luckily, we have an **add()**

method that can take care of this. We finally just need to "add" **orphan** to **root,** and we're done.

The **balance()** method uses a really simple algorithm to keep some semblance of balance in the tree structure. All we do is check that one branch is not more than one node larger than the other branch. If one branch is larger, we rotate in the opposite direction until the branches are within one node of each other. We put in the "one more" qualification because, if the total number of nodes in the branches is an odd number, it is impossible to get the branches even. Like many of the other methods in the tree, **balance()** is recursive. After we've balanced the subtree rooted at the **root** node, we recursively balance each branch in the same manner

As with the **rotate()** method, the Java code for **balance()** is relatively simple. First, we get the count for each branch, and then we rotate the imbalance in the opposite direction and get a new count. These steps are continued in a **while** loop until the branches are about even. Finally, we call **balance()** for the left and right child of the local **root** node and continue

down the tree until the leaf nodes are reached and the tree is reasonably balanced.

Now we'll test out this new tree, as shown in Figure 10-8, and see how close we come to a balanced tree. The **`TreeTest2`** class is the same as the **`TreeTest`** class. In this case, though, we are going to use a larger data set to demonstrate the balance functionality. For our data strings, we will just use the letters of the alphabet, A-Z.

Figure 10-8
TreeTest2 java.

```
package adt.Chapter10;

public class TreeTest2
{
      public static void main( String args[] )
      {
         BalTree t = new BalTree(
             new Comparable()
             {
                 public int compare( Object a, Object b )
                 {
                     return ((String)a).compareTo(
(String)b );
                 }
             }
         );

         t.add("A");
         t.add("B");
```


Figure 10-8
Continued.

```
         t.add("C");
         t.add("D");
         t.add("E");
         t.add("F");
         t.add("G");
         t.add("H");
         t.add("I");
         t.add("J");
         t.add("K");
         t.add("L");
         t.add("M");
         t.add("N");
         t.add("O");
         t.add("P");
         t.add("Q");
         t.add("R");
         t.add("S");
         t.add("T");
         t.add("U");
         t.add("V");
         t.add("W");
```

```
        t.add("X");
        t.add("Y");
        t.add("Z");

        t.traverse( Tree.PREORDER,
            new Traversal()
            {
               public void process(Object o)
               {
                   System.out.println( o );
               }
            }
         );
      }
}
```

In this test, we explicitly use the pre-order traversal to demonstrate the tree structure. By using pre-order, the data in the node is printed as soon as the node is accessed, before the move to the next node. This gives you an idea of how the tree is actually structured internally.

There are better algorithms that can be used to balance a binary tree.[1] In this example, the goal was to avoid the worst-case binary tree scenario of all the nodes being placed along one straight branch of the tree (like a

---

[1]Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill/MIT Press, New York, 1990.

linked list). The balancing algorithm presented here was kept simple to illustrate tree rotation. There is no reason why you could not implement a more complex (and efficient) algorithm for a better binary tree balance. Such a tree class easily could be subclassed from the **`BalTree`** class itself. Minimally, all the developer would need to do is to override the **`balance()`** method with a better implementation. The **`balance()`** method could adjust for the number of children on a branch, for example, as well as for how full each level of the tree is.

Exercises

1. In addition to the in-order and pre-order traversals covered here, there are other traversal algorithms. Modify the **`Tree`** class to support the following additional traversal types:

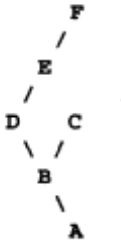
`POSTORDER`: The opposite of **`PREORDER`** It processes the left child, the right child, and the root.

`LEVELORDER:` Processes each level in full before processing the next level. (Hint: The recursive methods used so far emulate a stack. This ordered traversal might be better implemented using a queue.)

2. Create a small Java application to print the tree structure to the console, as shown in this example:

3. Create a small Java application to print the tree structure to the console from the leaves down, as shown in this example:

4. Create a small Java application to print the tree structure to the console sideways, as shown in this example:



5. Use the tree-printing methods used in the earlier exercises to demonstrate the changes in the tree structure due to the rotations performed in balancing the tree.

Summary

In this chapter, we learned the following:

- How to implement a binary tree.
- The binary tree node is very similar to the node used for a doubly-linked list.
- How to use recursive methods to make repetitive tree operations easier.
- How to quickly and easily search the binary tree structure.
- How to perform simple rotations within the tree structure to better balance the binary tree.

Chapter 11
Multi-Way Trees

This chapter explains the structure of more complex tree types. It expands on the binary trees

covered so far and takes a close look at a specific multi-way tree: the 2-3-4 tree. Here, we'll draw comparisons between the newly introduced multi-way trees and the binary tree structures we examined previously. Examples are provided to illustrate how a multi-way tree can be rendered as a binary implementation. We'll walk through implementations of the tree types in the examples. Exercises near the end of this chapter encourage us to develop other variations of multi-way trees.

Adding Complexity: Multi-Way Nodes

So far, the discussion of tree types has been restricted to binary trees. A *binary tree* is a specific implementation of the general tree structure in which the nodes of the tree have exactly two possible branches. There is no reason why a tree cannot be constructed with nodes that have a larger number of branches, though. This general type of tree can be called a *multi-way tree*.

A tree node can be constructed to contain two, three, four, or *n* number of branches. To support these branches, the node needs to support an appropriate number of keys. In the binary tree node, there is one key for two branches—left and right. The left branch is for nodes with keys less than the current key. The right branch is for nodes with keys greater than the current key.

In a tree that uses nodes that support three branches, the node needs to have two keys. The first key should be less than the second key. The left branch is for nodes with keys less than the first key. The center branch is for nodes with keys greater than the first key but less than the second key. The right branch is for nodes with keys greater than the second key.

A two-branch node requires one key. A three-branch node requires two keys. In the same fashion, a four-branch node requires three keys. As a matter of fact, an *n* branch node requires *n*-1 keys. There is always one less key than the number of branches the node supports; Figure 11-1 illustrates this point. The alphabetic characters in each node represent the keys the node contains, and the numbers represent the branches the node supports.

Sample I represents the typical binary (or two-way) node we've used before with its single key and two branches. Sample 2 represents a three-way node, which needs two keys to support its three branches. Sample 3 represents a four-way node with three keys and four branches.

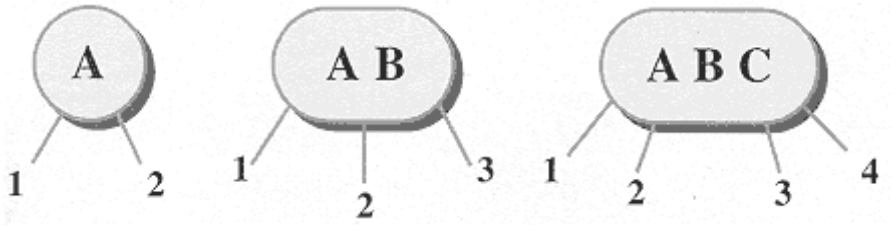


Figure 11-1
Multi-way nodes always have one more branch than the number of data items (or keys).

It is typical for multi-way trees to have nodes of several types. A tree that supports three-way

nodes, for example, generally will have both two-way and three-way nodes populating the tree, because it makes the tree easier to manage. A 2-3-4 tree will contain two-way nodes, three-way nodes, and four-way nodes.

2-3-4 Trees

A big difference between a multi-way tree like this and the previous binary tree is that, in a multi-way tree, data items always are added to the leaf nodes, and the tree grows up instead of down. A 2-3-4 tree is a tree structure that contains two-way, three-way, and four-way nodes—just as its name implies. The tree starts out with a two-way node and one data item. As data is added to the tree, the node fills up. The tree first becomes a three-way node and then a four-way node. When it reaches its maximum capacity of three data items, the tree splits into three 2-way nodes; the middle node is pushed up one level, and the right and left nodes become its right and left children. In this case, the tree creates a new level in the process. The new data item is placed in the appropriate two-way node, thereby making it a three-way node. The process continues this way as more data items are added to the tree. The new data always is added to the appropriate leaf node, though. All the leaves for the tree always are at the same level.

Now walk through the population of a small 2-3-4 tree to get a better idea of how it works. We'll start with an empty tree and add the sequence of data C-H-M-U-A-F-D. See Figure 11.2 for a representation of the tree after each step.

1. Add C to the empty tree. This creates a two-way or binary node as the root of the tree. Because there is only one data item, there are two possible branches to the node at this point.

2. Add H to the root node. The root node becomes a three-way node with the data items in sorted order.

3. Add M to the root node. At this point, the node becomes a four-way node, with three data items and four possible branches.

4. Add U to the root node. At this point, the node already is full. To add the new data item, the node must be split. C and M each become two-way nodes. H is pushed up one level on the tree to create a new root level. H is a two-way node that parents both the C

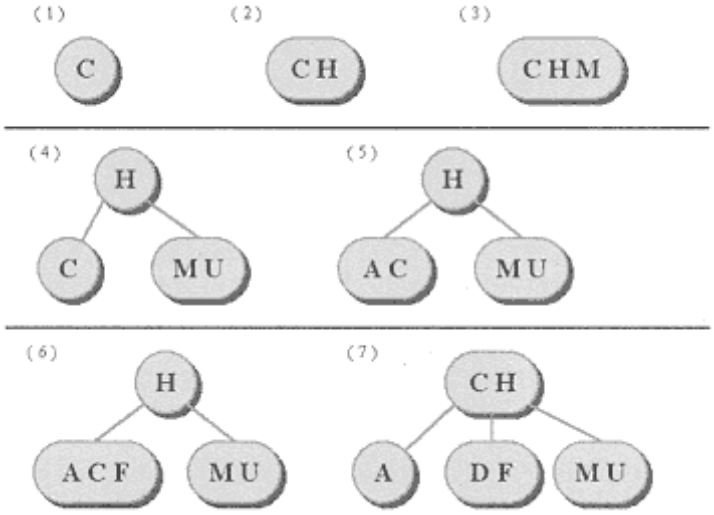


Figure 11-2
A 2-3-4 tree structure grown by
adding the sequence of data C-H-M-U-A-F-D.

and M nodes. Finally, the U data item is added to the appropriate two-way node, and M promotes the two-way M to the three-way M-U.

5. A is added to the tree using the same kind of placement algorithm as used in the binary tree. It compares as less than H, so it is assigned to the left child of the root node. Because the left child is a two-way node, it is promoted to a three-way node and now holds both the A and C data items.

6. F is added to the tree next. It compares as less than the root data item H, so it is assigned to the left child A-C. The child node already is a three-way node, so it is promoted to a four-way node containing A, C, and E

7. Finally, D is added to the tree. Again, it compares as less than H and is assigned to the left child. This time, though, the left child is already a four-way node. It needs to be split again. As before, the two outer data items are populated into two-way nodes, and the middle item, C, is pushed up one level in the tree. The two-way node containing H is already at the root level, so the C data item is added, which creates a three-way node at the root. Finally, the correct two-way node is identified and D is added, which creates the three-way node containing D-F.

One thing that should stand out while looking at the tree representations in Figure 11-2 is that the 2-3-4 tree always is extremely well balanced.

This is a consequence of the structure of the tree combined with the method by which the new data objects are added to the tree. New nodes are split from old nodes as space is needed, but the tree only grows new levels from the root end of the structure.

The Red-Black Tree: A Binary Version of the 2-3-4 Tree

As we can ascertain from the discussion so far, the 2-3-4 tree is a fairly efficient model for

data storage. These trees are self balancing and require no additional work by the user in comparison to binary trees. The 2-3-4 tree also is a fairly complex structure to implement. Instead of implementing that structure here, we'll take a look at 2-3-4 trees in a different fashion—one that keeps the benefit of the balanced tree and basic 2-3-4 structure but is much easier to implement.

It is possible to represent the three-way and four-way nodes as a combination of two-way nodes. Take a look at the four-way nodes in Figure 11-3. The node on the left is a traditional four-way node, like the one we've looked at so far. On the right is the two-way node equivalent of the four-way node. This binary equivalent will be called a four-way cluster. We can see that the binary representation retains all the important properties of the four-way node. There are still three data objects maintaining four external branches. The binary representation does have one major difference from the traditional four-way node: It requires an extra level to provide the same characteristics.

In the same way, a three-way node can be represented by two 2-way nodes in a cluster There are two data items and three external branches to the binary representations just like the traditional equivalent. Once again, the price we pay for this convenience is measured in additional levels to the tree.

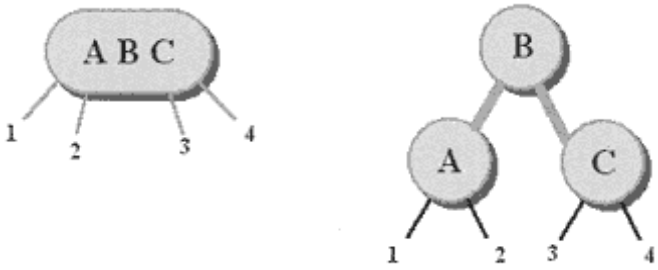


Figure 11-3
A traditional 4-way node can be represented by a cluster of 2-way nodes.

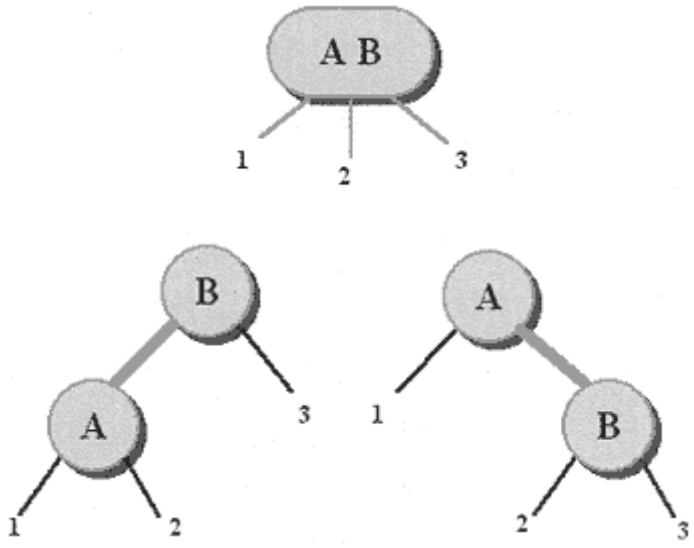


Figure 11-4
Binary representations of a 3-way node.

In both Figures 11-3 and 11-4, you will notice that some of the branches are thicker lines than the others. These thicker lines represent the internal links of the multi-way node. In the case of the four-way node, the link between A and B and the link between B and C symbolize the relationship of the data objects within the four-way node. The same goes for the three-way node; the link between A and B is internal to the three-way node cluster.

Traditionally, in red-black tree diagrams, the internal links are red and the external links are black. This modified tree type gets its name from these diagrams: It is called a *red-black tree*. When we move on to the implementation of the red-black tree, we will use a boolean value to denote the color of the link: **`true`** for red and **`false`** for black. A node's color is considered to be that of its link with its parent. The color differentiation is the key to the maintenance of the tree's balance. We use this information with two rules to maintain balance:

- There can never be two consecutive red links on a branch.
- The number of black links in the path between each leaf and the root node should be equal on all branches.

The rules are pretty easy to understand. Because the red links represent an internal relationship, when would you ever want two reds in a row? Whenever an operation performed on the tree violates one of these rules, a rotation is performed to correct the imbalance.

Implementing a Red-Black Tree

Now take a look at the implementation for the red-black tree. The tree will require interfaces for **`Comparable`** and **`Traversal`**, just like the binary tree. Figures 11-5 and 11-6 list these two interfaces. They are identical to the interfaces in the binary tree.

The red-black tree also will need a node class to hold the data. This node will provide quite a bit more functionality than the nodes defined previously. Figure 11-7 shows the complete source listing for the **`TreeNode`** class for the red-black tree.

Figure 11-5
Comparable.java.

```
package adt.Chapter11;

public interface Comparable
{
      public int compare( Object a, Object b );
}
```

Figure 11-6
Traversal.Java.

```
package adt.Chapter11;

public interface Traversal
{
      public void process( Object o );
```

```
}
```

Figure 11-7
TreeNode.java.

```
package adt.Chapter11;

public class TreeNode
{
	public TreeNode( Object o )
	{
		data = o;
		color = RED;
		left = right = null;
	}

	public Object getData()
	{
		return data;
	}
```

Continues

Figure 11-7
Continued.

```
	public void setData( Object o )
	{
		data = o;
	}

	public TreeNode getLeft()
	{
		return left;
	}

	public void setLeft( TreeNode l )
	{
		left = l;
	}

	public TreeNode getRight()
	{
		return right;
	}

	public void setRight( TreeNode r )
	{
		right = r;
	}

	public boolean getColor()
	{
		return color;
	}
```

```
public void setColor( boolean c )
{
      color = c;
}

public void flip()
{
      color = !color;
}

public boolean hasRedChild()
{
      if( left != null && left.color == RED )
            return true;

      if( right != null && right.color == RED )
            return true;

      return false;
}
```

Continues

Figure 11-7
Continued.

```
public boolean is2Way()
{
      if( color == RED )
          return false;

      return !hasRedChild();
}

public boolean is3Way()
{
      if( color == RED )
          return false;

      if( is2Way() || is4Way() )
            return false;

      return true;
}

public boolean is4Way()
{
      if( color == RED )
          return false;

      if( left == null || right == null )
          return false;

      if( left.color == RED && right.color == RED )
```

```
                return true;

            return false;
        }

        public String toString()
        {
            return "Node " + data;
        }

        private TreeNode left;
        private TreeNode right;
        private Object data;
        private boolean color;

        public static final boolean RED = true;
        public static final boolean BLACK = false;
}
```

In this version of the **`TreeNode`**, we have added a few extra member fields. First, we needed to add a field to represent the color of the node. The earlier explanation described the links between the nodes as having

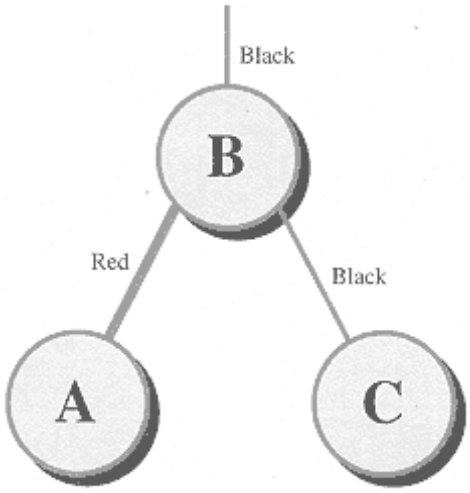


Figure 11-8
A node inherits the "color" of its parent branch.

a color. For all practical purposes, the color may safely be associated with the node to represent the link between the node and its parent. In the example in Figure 11-8, the B node is the root. The tree root implicitly has a black link, so the node's color is black. The A node has a red link to the B node, making the A-B cluster a three-way node. Because the link from the A child to the B parent is red, the A node carries a red color. The B-C link is black. C, in this case, is a two-way node attached to the A-B three-way cluster node. The link is black, so the C node is black.

In addition to the new boolean **`color`** field are the public final static ints **`RED`** and **`BLACK`** We have arbitrarily assigned **`True`** to **`RED`**, and **`false`** has been assigned to **`BLACK`** In the

`TreeNode` constructor, the color of the node is set to **`RED`**. With the exception of the tree's root node, every node added to the red-black tree will be added as a **`RED`** link to an existing node. Remember that, in the examples in the traditional 2-3-4 tree, adds always occur in an existing two- or three-way node. If a four-way node is the intended insertion point, it is split into two 2-way nodes prior to the add operation.

The **`TreeNode`** class contains the same accessor methods the binary tree did: **`getLeft()`**, **`setLeft()`**, **`getRight()`**, **`setRight()`**, **`getData()`**, and **`setData()`**. In addition to these accessors, a few new methods are available in the **`TreeNode`** for the red-black tree.

The **`getColor()`** and **`setColor()`** methods are standard accessors for the **`color`** field. Also, a utility method called **`flip()`** sets the **`color`** field. As its name implies, **`flip()`** is used to flip the color of the node from **`BLACK`** to **`RED`** or **`RED`** to **`BLACK`**. There is one more method in the **`TreeNode`**

class that is used specifically with color operations. The **`hasRedChild()`** method can be used to determine whether either of the children of the node is **`RED`** linked. The capability to determine whether a **`RED`** child exists, without specifically resolving which child it is, is very handy in some of the operations performed internally to the node and externally on the red-black tree itself We'll look at some of these operations next.

Because the red-black tree is a binary representation of a multi-way tree, it sometimes is desirable to know exactly what kind of multi-way node is being modeled in the binary format. The next three methods address this issue. Before we look at the individual methods, remember that it is possible to determine the node configuration only from the root node of any particular model. If a node has a red link to it, it is a child in a configuration and cannot determine the type of its parent. So these three methods are valid only on black nodes.

A node can have six basic configurations if we disregard direction:

1. No children

2. Two black children

3. Two red children

4. One black and one red child

5. One red child only

6. One black child only

These configurations are without regard to direction, because we don't particularly care in this determination which child is which color. It is only relevant that a child or the specified color exists—left or right.

The **`is2Way()`** method determines whether a particular node is a two-way node cluster. It first checks to see that it is not a red node, as explained earlier. Of the six configurations here, numbers 1 and 2 represent two-way nodes. Programmatically, this can be determined simply by checking that there are no red children. So the bottom line is that if a node is not red and has no

red children, it is a two-way node.

The **`is4Way()`** method checks to see whether this node is the root of a completely filled four-way node cluster. The only way a node can be a four-way node is if the node itself is black and both its children are red. Notice that in Figure 11-7, before attempting to access the **`color`** field of each child, we first determine whether the child exists. One of the most common mistakes made by novice programmers is trying to access the fields or methods of an object that doesn't exist. Skipping the check for existence leads to a lot of **`NullPointerExceptions`**.

All black nodes will be roots of clusters that will fall into one of three categories: two-way, three-way, or four-way clusters. This makes the implementation of the **`is3Way()`** method very simple. If a node is black, and it is not a two-way or a four-way node, it must be a three-way cluster.

Finally, the node class supplies the **`toString()`** method so that printing the node is equivalent to printing the **`data`** field only. It also could be used to print the node's color for debugging purposes.

Next comes the tree class itself We defined the **`RBTree`** class to implement the red-black tree; Figure 11-9 shows the complete listing. The public method names are the same as in the **`Tree`** class, but the implementations are substantially different.

Figure 11-9
RDTree.java.

```
package adt.Chapter11;

public class RBTree
{
      public RBTree( Comparable c )
      {
            this.c = c;
            root = null;
      }

      public void add( Object o )
      {
            root = add( root, new TreeNode(o) );
            root.setColor( TreeNode.BLACK );
      }

      protected TreeNode add( TreeNode root, TreeNode
            newNode )
      {
            if( root == null )
                  return newNode;

            if( root.is4Way() )
                  split(root);

            int val = c.compare( newNode.getData(),
```

```
                root.getData() );

        if( val < 0 )
        {
                if( root.getLeft() == null )
                {
                        root.setLeft( newNode );
```

Continues

Figure 11-9
Continued.

```
                }
                else
                {
                        root.setLeft(add( root.getLeft(),
                                newNode ));
                }
        }
        else
        {
                if( root.getRight() == null )
                {
                        root.setRight( newNode );
                }
                else
                {
                        root.setRight(add( root.getRight(),
                                newNode ));
                }
        }

        root =  balance( root );
        return root;
}

protected TreeNode balance( TreeNode node )
{
        if( node.hasRedChild() == false )
                return node;

        TreeNode child = node.getLeft();

        if( child != null )
        {
                if( child.hasRedChild() == true )
                        node = rotate( node, RIGHT );
        }

        child = node.getRight();

        if( child != null )
        {
                if( child.hasRedChild() == true )
```

```
                    node = rotate( node, LEFT );
            }

            return node;
      }
      public void split( TreeNode node )
```

Continues

Figure 11-9
Continued.

```
      {
            node.flip();
            node.getRight().flip();
            node.getLeft().flip();
      }

      protected TreeNode rotate( TreeNode root, int
            direction )
      {
            TreeNode newRoot = null;
            TreeNode orphan = null;
            boolean tmp;

            tmp = root.getColor();
            switch( direction )
            {
                  case RIGHT:
                      newRoot = root.getLeft();
                      root.setLeft( null );
                      orphan = newRoot.getRight();
                      newRoot.setRight( root );
                      break;
                  case LEFT:
                      newRoot = root.getRight();
                      root.setRight( null );
                      orphan = newRoot.getLeft();
                      newRoot.setLeft( root );
                      break;
            }

            if( newRoot == null )
                  return root;

            root.setColor( newRoot.getColor() );
            newRoot.setColor( tmp );
            if( orphan != null )
                  add( root, orphan );

            return newRoot;
      }

      protected Object search( TreeNode root, Object o )
      {
```

```
        if( root == null )
        {
             return null;
        }

        int val = c.compare( o, root.getData() );

        if( val == 0 )
        {
```

Continues

Figure 11-9
Continued.

```
            return root.getData();
        }
        else if( val < 0 )
        {
            return search( root.getLeft(), o );
        }
        else if( val > 0 )
        {
            return search( root.getRight(), o );
        }
        return null;
    }

    public void traverse( Traversal t )
    {
        traverse( INORDER, t );
    }

    public void traverse( int type, Traversal t )
    {
        traverse( root, type, t );
    }

    protected void traverse( TreeNode root, int type,
         Traversal t )
    {
        TreeNode tmp;

        if( type == PREORDER )
            t.process( root.getData() );

        if( (tmp = root.getLeft()) != null )
            traverse( tmp, type, t );

        if( type == INORDER )
            t.process( root.getData() );

        if( (tmp = root.getRight()) != null )
            traverse( tmp, type, t );
    }
```

```
        protected TreeNode root;
        protected TreeNode lastBlack;
        protected Comparable c;

        public final static int INORDER = 1;
        public final static int PREORDER = 2;

        protected final static int RIGHT = 1;
        protected final static int LEFT = 2;
}
```


The constructor for the **RBTree** class is the same as the constructor for the **Tree** class. It takes an object of type **Comparable** as an argument. This is used to perform comparisons between the data stored in the nodes.

The public **add()** method in **RBTree** takes an object as an argument just as its **Tree** counterpart. It also calls the protected version of the **add** (method in turn. Additionally, in this version of the method, we need to call **setColor()** on the root node to make sure that it stays black. It is important to keep the root node black to correctly balance the tree. Remember that, when splitting a four-way cluster, the colors of the nodes flip. If the root node splits, it leaves the root as a red node. Red denotes that the node is a child in a three-way or four-way node. The root of the tree by definition cannot be a child of anything, so we artificially force the root to stay black at all times.

The protected **add()** method in the **RBTree** class is where a lot of the action in the tree takes place. As with the **Tree** class, the protected **add()** method takes two arguments-both of the **TreeNode**'s**.** The first is the **root** of the subtree to which this node is being added. The second is the **newNode** that is being added to the tree. As before, if the **root** passed to the method is **null,** the tree is empty and the node becomes the new root of the tree. If this is the first node added to the tree, it is added as a red two-way node. After returning to the caller (the public **add()** method), the node is converted to a black two-way node. This is the only case in which a node is added as a two-way node.

If the tree already is partially populated, we need to add the node in the appropriate place and then make sure that the tree still is balanced. In the description of the traditional 2-3-4 tree, we discussed that the target for the new node is identified and then, if necessary, the target is split. If the target is a four-way node, it needs to be split into two 2-way nodes before the new node can be added. Splitting the four-way node pushes a two-way node up one level in the tree. This push could cause another four-way node to split, and so on. The normal processing on a 2-3-4 tree is a two-pass operation. The new node travels down the tree to its destination, and then the changes to the tree structure are propagated up the tree level by level until the root is reached.

In the red-black tree, we take a slightly different approach. Because the **add()** method uses recursion, we still have what amounts to a two-pass operation: one pass while recursing deeper into the tree structure and another as we come out of each layer of the recursion. In our method, though, we'll do all the splitting on the first pass on the way down the tree to find the new node's target. This is perfectly legal and acceptable,

because we will pass through only node clusters that are along the path to the new target position in the tree. It is quite likely that we'll end up having to split any four-way node clusters we pass through anyway. So, after checking whether we are adding the root node to the empty tree, we immediately test to see whether the current node is the root of a four-way cluster. If it is, we split the node at this point.

The next step is to determine the branch to which this node belongs. We perform the comparison and check for a value less than zero. If it is less than zero, we know that the new node will go somewhere down the left branch of the current subtree. If there is no node on the left branch, we have found the target position for the new node and can assign it as the left child of the current root. If the branch is not empty, we need to recurse down the tree to the left one more level and start the process all over again. The process is the same for the right branch.

Note that the recursive call to the **`add()`** method is embedded in a call to the **`setLeft()`** method. This is done because it is quite possible that the child we pass as the root to the new subtree in the recursive call may be rotated out of place on the return. As you can see, after the new node is added to the tree and right before we return from the recursive call, the method calls the **`balance()`** method to make sure that the tree structure still is intact.

The **`balance()`** method's sole purpose is to maintain the tree's red-black structure. This is accomplished by performing rotations on the subtrees as necessary. Under what conditions will we need to perform rotations? Earlier in this chapter, we learned that there can never be two consecutive red links on a branch. But the process of adding the new node gave no regard to the color of the nodes being processed. This inevitably leads to circumstances in which the add operation will break this rule. The **`balance()`** method solves the problem by rotating the double **`RED`** link into a legal red-black configuration.

Because the condition we need to correct in the **`balance()`** method is brought about by two consecutive red nodes on the same branch, we immediately can cease the operation if the current node has no red children. So **`balance()`** makes an immediate call to the node's **`hasRedChildren()`** method. If this call returns **`false`**, we're done. If the node does have at least one red child, we determine which child by testing each one. When we find a red child node, we check it to see whether it has a red child. This time, we use the child node's **`hasRedChild()`** method. If this call returns **`true`**, we've encountered an illegal configuration. To remedy the situation, we simply rotate the offending node in the opposite direction of the inappropriate red

child. If the node passed as an argument to the **`balance()`** method has a left red child that, in turn, has a red child, for example, we rotate right. If the argument node's right child is red, and that node has a red child, we rotate left.

These rotations have two effects. The first is that the subtree to which the new node originally was added now has a new root. This new root is returned from the **`balance()`** method to the **`add()`** method, which then returns the new root of the subtree affected to its caller (the recursive **`add()`**). This repeats along the path from the target subtree to the real root of the

whole tree structure, thereby supplying us with the second pass of our two-pass add operation. The rotations also automatically force the tree into a very balanced structure.

The **`split()`** method takes a **`TreeNode`** node as an argument. This node should be the root of a four-way cluster. If it is not a four-way cluster, the method returns immediately. As discussed earlier, the only thing that needs to be done to split a four-way cluster is to flip the colors of all the links or, in this case, the nodes. We don't need to worry about any of the children being **`null`**, because we've already established that this is a four-way cluster and therefore has two red children.

The **`rotate()`** method is the same for the **`RBTree`** as it was for the **`Tree`** class. The only difference is that there are a few statements in the method that maintain the colors of the **`root`** and **`newRoot`** nodes. The idea here is that, even though the nodes have been rotated into new positions, the colors of the links between the nodes should not change. If a pair of nodes, A and B, have a red link with A as the parent, rotation around B causes B to become A's parent instead of its child. To maintain the relationship of the two nodes (A-B being red), we need to swap the colors of the nodes. This also has the effect of the former relationship between A and its parent being maintained by B as the new root of the subtree.

The **`RBTree`** versions of the **`search()`** and **`traverse()`** methods are identical to those found in the **`Tree`** class. And that covers the entire **`RBTree`** class. Next, we'll take our tree for a test drive.

Using a Red-Black Tree

The **`RBTreeTest`** class is a rehash of the **`TreeTest`** worst-case scenario for a binary tree. All the data is added to the tree already in sort order. With a data set like this, we expect that the tree would become lopsided to the

right. Fortunately, the red-black tree takes care of this problem, and we end up with a reasonably balanced tree in any case. Figure 11-10 shows the source code for the **`RBTreeTest`** class.

Figure 11-10
RDTreeTest.java.

```
package adt.Chapter11;

public class RBTreeTest
{
      public static void main( String args[] )
      {
         RBTree t = new RBTree(
             new Comparable()
             {
                 public int compare( Object a, Object b )
                 {
                     return ((String)a).compareTo(
                           (String)b );
                 }
             }
```

```
        );

        t.add("A");
        t.add("B");
        t.add("C");
        t.add("D");
        t.add("E");
        t.add("F");
        t.add("G");
        t.add("H");
        t.add("I");
        t.add("J");
        t.add("K");
        t.add("L");
        t.add("M");
        t.add("N");
        t.add("O");
        t.add("P");
        t.add("Q");
        t.add("R");
        t.add("S");
        t.add("T");
        t.add("U");
        t.add("V");
        t.add("W");
        t.add("X");
        t.add("Y");
        t.add("Z");

        t.traverse( RBTree.PREORDER,
```

Continues

Figure 11-10
Continued.

```
            new Traversal()
            {
               public void process(Object o)
               {
                   System.out.println( o );
               }
            }
        );
    }
}
```

If you did the exercise in the last chapter to print the tree structure, you can plug that into the **`RBTree`** to see the final structure of the tree. Otherwise, you can use the **`PREORDER`** traversal output to visualize the tree structure as it would appear. The first node listed is the root of the tree. The nodes then follow, from the left branch down to the leaves. Then the right branches of the nodes are listed as far back to the root node. Finally, the process is mirrored for the right branch of the root.

Exercises

1. Create a small Java application to print the red-black tree structure to the console. You will need to be a little more creative than with the binary trees. Make sure to indicate the color of each branch/node in the output.

2. Use the tree-printing method in Exercise 1 to demonstrate the changes in the tree structure as data is added to the tree. Pay special attention to what happens to the structure when nodes are filled and split.

Summary

In this chapter, we learned the following:

- How multi-way trees are structured.
- How a 2-3-4 tree is different from a standard binary tree.
- How the structure inherent in the 2-3-4 tree forces the tree to be as balanced as possible at all times.
- How to use the binary tree structure to represent higher order trees, such as the red-black implementation of the 2-3-4 tree.
- How to split full nodes to make room for tree structure expansion.

Chapter 12
B-Trees

In this chapter, we'll take a detailed look at the B-Tree data structure as an extension of the red-black and 2-3-4 trees. B-Trees typically are used to index large data sets and external data stores, such as database files. In this chapter, we'll take a look at a simple B-Tree implementation to help us walk through the concepts presented here. Exercises near the end of this chapter include developing a simple indexed data file.

B-Trees

The red-black trees discussed in Chapter 11 are examples of multi-way trees of a fixed order.

The term *order*, in this context, describes the maximum number of branches a node can support. The 2-3-4 and red-black trees in Chapter 11 have an order of 4. In other words, there is a maximum of three data elements or keys and four branches per multi-way node. In this chapter, we'll discuss a multi-way tree structure with an arbitrary number of data items per node: a *B-Tree.*

As in all the previous tree examples, a B-Tree of order *N* will have nodes with a maximum of *N* branches and *N*-1 keys or data elements. Figure 12-1 illustrates a partially filled multi-way tree of order 5. The full root node has four key values and five branches with children. The first three children—A-B, D-E-F, and H-I—are partially filled. The last two children—K-L-M-N and Q-R-S-T—are filled completely. The numbers at the bottom of the illustration show the possible branches based on the current number of keys in each node. Because the tree has an order of 5, all nodes have the potential to parent five children on five separate branches. Because a node can support only one more branch than the number of keys contained, though, the partially filled nodes only show potential for the number of branches indicated by the number of keys the nodes currently have.

A B-Tree of order 5 with two completely full levels has 24 keys and potentially 25 branches. If a third level is filled, the tree holds 149 keys with 150 potential branches. One of the advantages of the B-Tree data type is that a large number of keys can be stored in a tree only a few levels deep. When the nodes are being stored and read from disk files, these larger nodes represent much more efficient access to the data. This is true especially when the node sizes are calibrated to match the block read size of the disk. Fewer nodes and levels in the tree also reduce the number of accesses required from the disk.

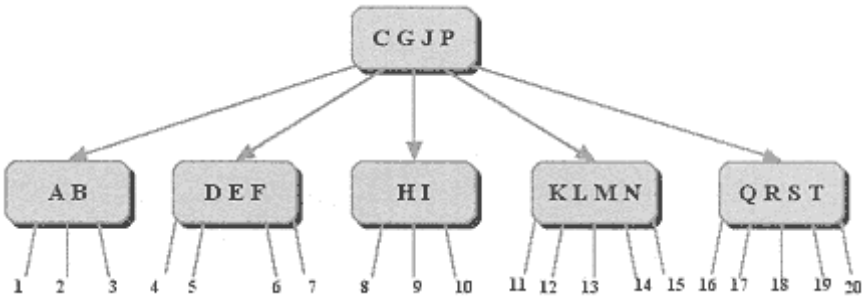


Figure 12-1
A partially filled 5-way tree.

One additional note about the number of keys and branches in a node: it is desirable to keep the nodes as full as possible in a B-Tree to promote efficiency. We therefore define a rule that requires that, at a minimum, half the node must be occupied at all times. In a B-Tree of order 6, for example, each node is required to have a minimum of three branches (order/2) and two keys (number of branches - 1).

Indexing Large Data Sets

One of the most common uses of the B-Tree data type is to represent an index for an *indexed sequential access method* (ISAM) database file. The data and index usually are kept in separate files. The index file allows for quick lookups in the data file by storing a unique key

along with a numeric offset in the data file to the beginning of the indexed record. The advantage of the index file is that, instead of having to sequentially search through the data file to find the desired record, an access program can look up the key in the much smaller index file to find the exact offset of the desired record within the data file. Not only is the index file inherently much smaller than the data file (because the key is usually a small part of the record), but it also is organized as a tree structure designed to optimize these kinds of lookups.

Generally, an application that uses a B-Tree type structure for an index optimizes the number of keys so that the size of each node is equal to the size of a page read in the native filesystem. By doing so, the application gets peak performance from the B-Tree, because the disk is read in its largest single-read blocks, which then correspond to the size of a node.

Node Width

Determining the correct node width is somewhat of a problem in Java. Because the optimum size of a B-Tree node is the number of keys that fit in a single disk page, a Java implementation of a B-Tree is at some disadvantage. Java is meant to be platform independent; that is one of the features that makes it so attractive as a development language. It also means that the developer is fairly well insulated from the native platform on which the code is run.

There is no way in Java to determine the page size of the local filesystem. There also is no easy way in Java to determine the size of an object. Therefore, any 100-percent Java-implemented B-Tree is not going to be able to optimize the number of keys per node to the disk page size. The best we can do is to use what seems to be a reasonable number of keys per node and count on the advantages of the tree storage structure to help give better performance. In any case, the performance of the B-Tree will be better than using a linear storage mechanism.

B-Tree Operations

Most of the operations performed on a B-Tree are very similar to those performed on any tree type. In operations such as tree traversal and searching, the only difference is in navigation. When performing a search, the process is basically the same as it is in the binary tree.

Searching a B-Tree

The search starts with the first key in the root node of the B-Tree. If the key matches the target, the search is complete. If the search target compares as less than the key, the search continues down the first branch. If the target compares greater than the key, the second key is tested. The comparison continues with the second key exactly as the first. The move to the second key is the equivalent of the move down the right branch in a binary tree. Each key is checked until one of three things happens:

- A match is found, which terminates the search.
- The target compares as less than the comparison key, which causes the search to proceed down the corresponding branch.

- If the target is greater than the last key in the node, the search continues down the right branch of the node.

The search continues in exactly the same manner for each node along the search path until the key matches the target or the search leads to a **`null`** branch, which indicates that the target does not match any key in the tree.

Traversing a B-Tree

The traversal operation predictably follows the same basic schema as the search operation. Assuming an in-order traversal, the tree is traversed branch - key - branch - key - branch, in much the same way as a binary tree. In a B-Tree, however, many of the branches are internal to the node. This section walks us through the basics of the B-Tree in-order traversal.

The traversal starts, as always, with the root node. An in-order traversal starts with the first branch. If the branch is not **`null`**, the traversal moves to the child node on the branch. If the branch is **`null`**, the first key is processed with the process method. Then the traversal moves to the second branch of the node. The traversal continues in this fashion until the last key in the node is processed, and the traversal moves to the child node on its right branch, if any.

As the traversal moves to the children on each branch, the process is repeated as though the child were the root of the tree. Because the traversal operation is recursive, after the right branch is completely processed, the method returns to allow the next level up to continue. The processing continues in the same way until every node on the tree is completely visited and processed.

Adding Keys to a B-Tree

The add operation for a B-Tree is similar to the add operation for a 2-3-4 tree. The location for the add is determined in exactly the same way as in a 2-3-4 tree. The tree structure is traversed looking for the node where the new key belongs. After the correct node is found, the key is inserted into the node in the proper location. There is one big difference in a B-Tree add operation compared to a 2-3-4 tree add. Unlike a 2-3-4 tree operation, a B-Tree add does not split full nodes automatically as it passes through searching for the insert location. In a B-Tree, we want to have the nodes as full as possible, which leads to the fewest number of overall nodes and levels.

Splitting the Nodes of a B-Tree

The process of splitting a B-Tree node is again very similar to splitting the nodes of a 2-3-4 tree. When a node is full and a new key needs to be inserted, the node first is split into two nodes. The process of splitting a

B-Tree node involves first determining the center key in the series. This middle key is pushed up to the next higher level in the tree, and it becomes a member of its former parent node. The remaining keys and all the branches from the original node are used to populate the new nodes.

The first branch in the node, the left branch, becomes the left branch of the new left node. The first key in the original node becomes the first key in the new left node. The same process follows for the second branch and key, the third branch and key, and so on until the middle of the node is reached.

Suppose that we are splitting a node with an order of 6, such as the one shown on the left in Figure 12-2. The first and second branch and key are transferred to the new left node. The third branch (of six total) becomes the new left node's right branch. The third key is promoted to the next higher level node, which, in turn, may cause that node to need splitting. The fourth branch of the original node becomes the left branch of the new right node. The fourth key in the original becomes the first key in the new right node. The fifth branch and key from the original become the second branch and key of the new right node, respectively. The sixth and final branch of the original node becomes the right branch of the new right node. We end up with the configuration shown on the right in Figure 12-2.

The key that was promoted to the parent of the original node now becomes the parent of the new left node. In other words, if the promoted key becomes the third key in the parent, the new left node is the child on the third branch of the parent. The branch that the original, unsplit node resided on ends up pointing to the new right node.

It is important to note that, although a 2-3-4 tree splits any full nodes encountered during an add operation, a B-Tree generally waits until an overflowing key is being added to the full node itself. With a B-Tree, we

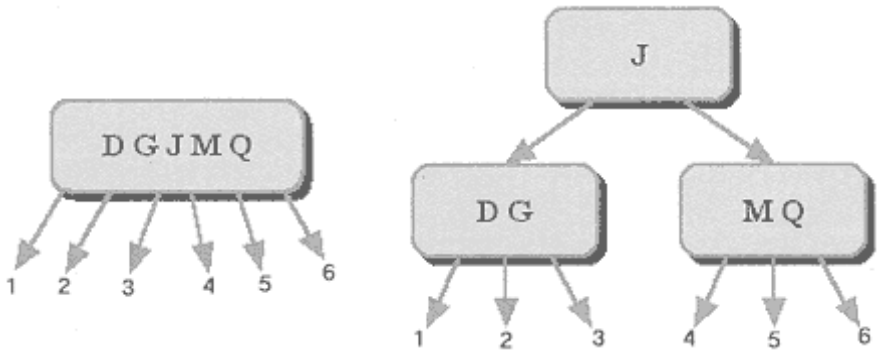


Figure 12-2
Splitting a 6-way node.

are looking to get as many full nodes as possible to keep the tree height to a minimum. If a B-Tree implementation is going to be used as a file-based index, for example, each node can be read from the index file as a single operation. Therefore, full nodes reduce the total number of read operations to load the tree.

Balancing a B-Tree

A B-Tree is pretty much a self-balancing construct. Because of the nature of the add and split operations, there generally is no need for any additional balancing after these operations. Deleting a key from a node can throw the tree out of balance, however. Earlier, we defined a rule stating that a node must be, at a minimum, half full. Deleting a key from a node can put it

under the minimum. In a case like this, we would rebalance the tree by rotation.

Representing a B-Tree with Binary Nodes

The B-Tree structure can be implemented in many ways. In this section, we'll implement our version of the B-Tree by using a binary representation of the multi-way nodes similar to the red-black representation of a 2-3-4 tree.

As with a red-black tree, it can be easier to work with a B-Tree structure by using a binary representation of the multi-way nodes. The biggest problem with the implementation is keeping track of which of the branches of the binary tree represent internal links between keys and which are external links to other multi-way nodes.

Compare the binary B-Tree representation in Figure 12-3 with its traditional representation in Figure 12-1. These two illustrations show the structure for the exact same tree. In the binary version, the thicker arrows marked with Rs (red) denote internal links between keys in the same node. The thinner arrows marked with Bs (black) signify external branches for the multi-way node. The black branches in Figure 12-3 correspond to the branches shown in Figure 12-1. The binary nodes C, G, J, and P are connected by red branches. These binary nodes, in reality, are siblings in the represented multi-way node of the B-Tree.

Page 222

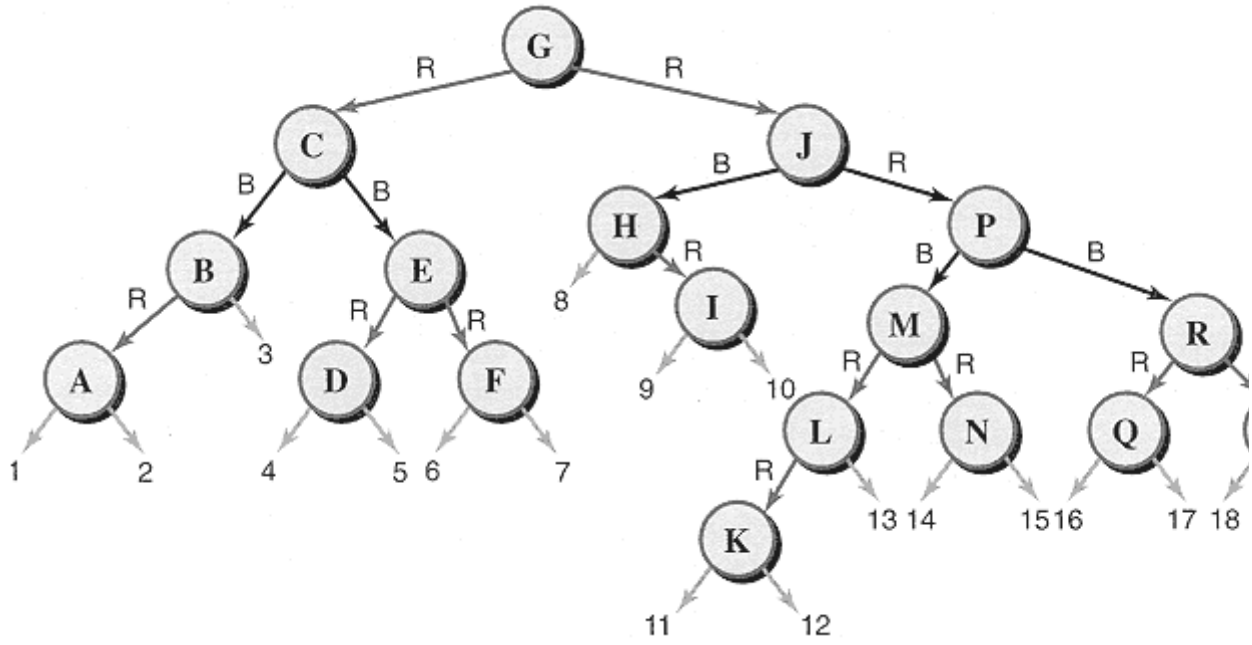


Figure 12-3
A binary represenation of a B-Tree.

Page 223

One rule that is not going to carry over from the red-black 2-3-4 tree is that the structure of a red-black tree forbids the occurrence of two consecutive red links. In the binary B-Tree, consecutive red links are common. The limit to consecutive red branches in the B-Tree is actually two less than the order of the tree. The maximum is based on the fact that there is one

less branch joining the multi-node siblings than there are siblings. It also has been established that there is, at a maximum, one less sibling than there are external branches (the number of which define the order).

Implementing a Binary B-Tree

In our binary implementation of a B-Tree, we need the same **`Comparable`** and **`Traversal`** interfaces we used in the rest of the tree implementations. Figures 12-4 and 12-5 show the source code for these interfaces.

The node class we will use for the B-Tree also is very similar to the node from the red-black tree in Chapter 11. The **`TreeNode`** class here contains the same **`data`**, **`color`**, **`left`**, and **`right`** fields as its predecessor. The accessor methods **`getData()`**, **`setData()`**, **`getLeft()`**, **`setLeft()`**, **`getRight()`**, **`setRight()`**, **`getColor()`**, and **`setColor()`** also are reused in this incarnation as is the utility method **`flip()`**.

In the B-Tree's **`TreeNode`** class, we maintain the concept of node color as the color of the branch from the parent (see Figure 12-6). Therefore, we retain the static final **`RED`** and **`BLACK`** boolean constants. Just like in the red-black tree, we will depend heavily on the color of the node to determine behavior during the processing of the operations on the B-Tree.

Figure 12-4
Comparable.java.

```
package adt.Chapter12;

public interface Comparable
{
      public int compare( Object a, Object b );
}
```

Figure 12-5
Traversal.Java.

```
package adt.Chapter12;

public interface Traversal
{
      public void process( Object o );
}
```


Figure 12-6
TreeNode.java.

```
package adt.Chapter12;

public class TreeNode
{
      public TreeNode( Object o )
      {
            data = o;
            color = RED;
```

```
        left = right = null;
    }

    public Object getData()
    {
        return data;
    }

    public void setData( Object o )
    {
        data = o;
    }

    public TreeNode getLeft()
    {
        return left;
    }

    public void setLeft( TreeNode l )
    {
        left = l;
    }

    public TreeNode getRight()
    {
        return right;
    }

    public void setRight( TreeNode r )
    {
        right = r;
    }

    public boolean getColor()
    {
        return color;
    }

    public void setColor( boolean c )
    {
        color = c;
    }

    public void flip()
    {
```


Figure 12-6
Continued.

```
        color = !color;
    }

    public int countRedChildren()
    {
        int count = 0;
```

```
            if( left != null && left.color == RED )
            {
                  count += left.countRedChildren();
                  count++;
            }

            if( right != null && right.color == RED )
            {
                  count += right.countRedChildren();
                  count++;
            }

            return count;
      }

      public boolean isOverFull(int order)
      {
            if( color == RED )
                return false;

            if( countRedChildren() >= order - 1 )
                return true;

            return false;
      }

      public String toString()
      {
            return "Node " + data;
      }

      private TreeNode left;
      private TreeNode right;
      private Object data;
      private boolean color;

      public static final boolean RED = true;
      public static final boolean BLACK = false;
}
```

Two new methods are present in the B-Tree's node class: **`countRedChildren()`** and **`isOverFull()`**. The **`countRedChildren()`** method is used to determine how many siblings are in the multi-node. For this implementation, the black node in any multi-node grouping is

considered to be the conceptual owner of the node. The reason for this is that, for a multi-node to exist, at least one key or data item must reside in one node. The black owner **`TreeNode`** object theoretically could have no red siblings (children on the binary tree), but it is impossible to have a sibling without its black owner **`TreeNode`** object. Also, because black represents an external connection, the parent of a black node is always from a different multi-node.

The **countRedChildren()** method is a recursive method that counts the number of red children the node has on each of its two branches. The recursion continues down each branch until a black child or no child is found. Remember that a black child means a separate multi-node, so it cannot be counted as a sibling. After both branches are counted in this manner, the total is returned to the calling method. As the recursion unwinds, the final return value contains a total of all red siblings for the multi-node.

The other new addition to the **TreeNode** class is the **isOverFull()** method. This method is used to determine whether the multi-node needs to be split. Only the owner node for the multi-node can make the determination of whether the group needs to be split. For this reason, the first thing the method does is check the color of the node. If the color is red, the method returns immediately. Otherwise, if it is a black node, the method checks the number of red children it has. If the number of red children is greater than or equal to one less than the order, the node is too full.

A B-Tree takes a different approach than a red-black tree when determining when a node needs to be split. This binary implementation takes the standard B-Tree approach one step further. In normal B-Tree processing, the system determines that a new key must be added to a full node. The node then is split according to the procedure outlined earlier, and the new key is added to the appropriate node.

In the binary implementation, we do the splitting after the key (new binary node) is added. This is covered in more detail shortly when we take a look at the B-Tree add operation. For now, it is important only to understand that we will be looking for the condition of an overfilled node. The method returns **true** or **false** based on this determination.

Now take a look at the **BTree** class itself in figure 12-7. Once again, this is a binary representation of a multi-way tree just like the red-black tree. The implementation shares many similarities with the red-black tree. The **BTree** also will have significant differences from the red-black tree implementation.

One difference is in the constructor. The **BTree** class constructor takes two arguments—the tree's order and a **Comparable** object—so that we can compare nodes.

A B-Tree has the same basic public API as a red-black tree. There is a public **add()** and a protected **add()** method. The public **add()** method is the same for a B-Tree as it is for a red-black tree. The protected **add()** method is a little different, however. As with all the binary trees, we first check to see whether the tree is empty. If it is, we assign the **newNode** to the root node by using **newNode** as the return value for the method. After we determine that this is not an empty tree, we need to determine whether the new node is greater than or less than the local root node.

If the new node is less, it goes somewhere on the left branch. If there is no left child of this subtree, the new node becomes the left child of the root node. If there is a left child, we recursively call **add()** with the left child as the root of the new add operation. If the new node is greater than the root node, the same process follows for the right branch.

So far, this process is similar to the red-black add operation. The one difference is that we

completely skipped the red-black's check for a full node and the subsequent split of the multi-node if, in fact, the node was full. A B-Tree has different requirements for splitting the multi-nodes. We don't want to split a full multi-node unless we need to actually add a new key to it. This presents us with a bit of a problem. If we attempt to split the node before adding the key, we most likely will end up needing to make two passes to add. The first pass identifies the multi-node into which the new key needs to be inserted. The target multi-node then is checked to determine whether it is full. If it is full, the node is split, and we can determine in which of the child nodes to insert the new key. The changes in the branch structure then are populated back up the tree via the return value from the recursive call. Then we can make a second pass to determine whether the tree has been thrown out of balance due to the add operation.

Suppose that we have a tree with an order of 5. The full node has four keys and five branches. To split the node, one of the keys must be promoted up one level to the parent multi-node. This leaves three keys to distribute between two nodes. One node gets two keys, and the other gets one. Suppose that the new key now needs to be added to the multi-node with the two keys. We end up with one child with three keys and the other with only one key. This is unbalanced, so we'll need to perform a rotation to rebalance the nodes.

As an alternative to this process, we are going to add the node first and then determine whether any splitting must be done. This greatly simplifies the implementation for us by eliminating the need to determine the target location for the new key a second time and the need to rebalance the tree across multiple multi-nodes.

So, to finish off the add operation, the method calls the **`balance()`** method and then checks for the need to split an overfilled multi-node. The **`split()`** method is called if necessary. The **`balance()`** method called here has nothing to do with rotating multi-nodes or even rotating keys and branches between multi-nodes. In this case, we need to make sure that the multi-node we are about to split is balanced internally. To simplify the split, we want the black owner binary node to be the node we will promote to the next level. It therefore needs to be in the middle of the multi-way node.

Figure 12-7
BTree java.

```
package adt.Chapter12;

public class BTree
{
	public BTree( int order, Comparable c )
	{
		this.order = order;
		this.c = c;
		root = null;
	}

	public void add( Object o )
	{
		root = add( root, new TreeNode(o) );
		root.setColor( TreeNode.BLACK );
	}
```

```
protected TreeNode add( TreeNode root, TreeNode
      newNode )
{
      if( root == null )
            return newNode;

      int val = c.compare( newNode.getData(),
            root.getData() );

      if( val < 0 )
      {
            if( root.getLeft() == null )
            {
                  root.setLeft( newNode );
            }
            else
            {
                  root.setLeft(add( root.getLeft(),
                        newNode ));
            }
      }
```

Continues

Figure 12-7
Continued.

```
      else
      {
            if( root.getRight() == null )
            {
                  root.setRight( newNode );
            }
            else
            {
                  root.setRight(add( root.getRight(),
                        newNode ));
            }
      }

      root =  balance( root );

      if( root.isOverFull(order) )
      {
            split(root);
      }

      return root;
}

protected int branchCount( TreeNode child )
{
      if( child != null )
      {
```

```
                if( child.getColor() == TreeNode.RED )
                {
                        return child.countRedChildren() + 1;
                }
        }
        return 0;
}

protected TreeNode balance( TreeNode node )
{
        if( node == null )
                return node;

        if( node.getColor() != TreeNode.BLACK )
                return node;

        if( node.countRedChildren() < 2 )
                return node;

        while( branchCount( node.getLeft() )
                < branchCount( node.getRight() ) - 1 )
        {
                node = rotate( node, LEFT );
        }
```

Continues

Figure 12-7
Continued.

```
        while( branchCount( node.getRight() )
                < branchCount( node.getLeft() ) - 1 )
        {
                node = rotate( node, RIGHT );
        }
        return node;
}

public void split( TreeNode node )
{
        if( node.isOverFull(order) == false )
                return;

        node.flip();
        if( node.getRight() != null )
                node.getRight().flip();
        if( node.getLeft() != null )
                node.getLeft().flip();
}
protected TreeNode rotate( TreeNode root, int
        direction )
{
        TreeNode newRoot = null;
        TreeNode orphan = null;
        boolean tmp;
```

```
            tmp = root.getColor();
            switch( direction )

            {
                  case RIGHT:
                        newRoot = root.getLeft();
                        if( newRoot == null )
                              return root;
                        orphan = newRoot.getRight();
                        root.setLeft( orphan );
                        newRoot.setRight( root );
                        break;
                  case LEFT:
                        newRoot = root.getRight();
                        if( newRoot == null )
                              return root;
                        orphan = newRoot.getLeft();
                        root.setRight( orphan );
                        newRoot.setLeft( root );
                        break;
            }

            root.setColor( newRoot.getColor() );
            newRoot.setColor( tmp );

            return newRoot;
```

Continues

Figure 12-7
Continued.

```
    }

    protected Object search( TreeNode root, Object o )
    {
        if( root == null )
        {
            return null;
        }

        int val = c.compare( o, root.getData() );

        if( val == 0 )
        {
            return root.getData();
        }
        else if( val < 0 )
        {
            return search( root.getLeft(), o );
        }
        else if( val > 0 )
        {
            return search( root.getRight(), o );
```

```
        }
        return null;
    }

    public void traverse( Traversal t )
    {
        traverse( INORDER, t );
    }

    public void traverse( int type, Traversal t )
    {
        traverse( root, type, t );
    }

    protected void traverse( TreeNode root, int type,
        Traversal t )
    {
        TreeNode tmp;

        if( type == PREORDER )
            t.process( root.getData() );

        if( (tmp = root.getLeft()) != null )
            traverse( tmp, type, t );

        if( type == INORDER )
            t.process( root.getData() );

        if( (tmp = root.getRight()) != null )
```

Continues

Figure 12-7
Continued.

```
            traverse( tmp, type, t );
    }

    protected TreeNode root;
    protected Comparable c;
    protected int order;

    public final static int INORDER = 1;
    public final static int PREORDER = 2;

    protected final static int RIGHT = 1;
    protected final static int LEFT = 2;
}
```

The **`branchCount()`** method is used to test the number of children on a branch of the multi-node. It includes the child node in the count and is safe to call even if the child node is **`null`**.

The **`balance()`** method is used to internally balance a multi-node. To simplify the split

operation, it is desirable to have the black owner binary node in the middle of the multi-node. *Middle*, in this case, means that the same number of red child nodes should exist on each of the black node's two branches (or as close as is possible). The first thing the method does is check to see whether this is a valid attempt to balance the multi-node. The call is valid if the node passed to the method meets three criteria:

- It is not null.
- It is colored black.
- It has at least two red children (it makes no sense to attempt to balance zero or one child!).

The actual processing of the balance operation is relatively simple. If there are less nodes in the left branch than one less the number in the right branch, the tree is rotated to the left:

```
( branchCount( node.getLeft()  ) < branchCount(
     node.getRight() ) - 1 )
```

Otherwise, if there are less nodes in the right branch than one less the number of nodes on the left branch, the tree is rotated to the right:

```
( branchCount( node.getRight() ) < branchCount(
     node.getLeft() ) - 1 )
```


The new balanced root of the multi-node is used as the return value of the method.

The **`split()`** method for a B-Tree is basically the same as the **`split()`** method for a red-black tree, with one exception. The B-Tree version of the method first does a check to make sure that the split is valid by calling the node's **`isOverFull()`** method. The split is performed by flipping the color of the nodes. The original black node becomes red, thereby becoming a sibling in the original parent multi-node, and each of the red children becomes a black owner of its own multi-nodes.

The **`rotate()`** method is used to move the members of a multi-node around until the multi-node is balanced. Of course, the **`balance()`** method controls the number of times **`rotate()`** is called and the direction of the rotation.

To rotate a B-Tree, we get the binary child opposite the direction of rotation and assign it as the **`newRoot`** node. We then store a reference to the child of the **`newRoot`**, if any, that is in the *same* direction as the rotation. This orphan subtree is assigned as the new child of the original root on the branch recently vacated by **`newNode()`**. Next, the old root is set as the new child of the **`newRoot`** in the branch formerly occupied by **`orphan`.**

The last two methods for a B-Tree, **`search()`** and **`traverse()`**, are exactly the same as with any of the other binary trees. We saw these same implementations in all the binary-based trees we examined.

Figure 12-8 shows a quick program to populate the tree.

Figure 12-8
BTreeTest.java.

```
package adt.Chapter12;

public class BTreeTest
{
      public static void main( String args[] )
      {
         String keys[] = {
           "A", "B", "C", "D", "E", "F", "G", "H",
           "I", "J", "K", "L", "M", "N", "O", "P",
           "Q", "R", "S", "T", "U", "V", "W", "X",
           "Y", "Z", "ZA", "YA", "UA", "XA", "WA",
           "VA", "TA", "SA", "RA", "QA", "PA", "OA",
           "NA", "MA", "LA", "KA", "JA", "IA", "HA",
           "GA", "FA", "EA", "DA", "CA", "BA", "AA" };

         BTree t = new BTree( 6,
             new Comparable()
             {
```

Continues

Figure 12-8
BTreeTest.java.

```
                  public int compare( Object a, Object b )
                  {
                      return ((String)a).compareTo(
                           (String)b );
                  }
               }
             );

           for( int i = 0; i < keys.length; i++ )
           {
             t.add(keys[i]);
           }

           t.traverse( BTree.INORDER,
               new Traversal()
               {
                  public void process(Object o)
                  {
                      System.out.println( o );
                  }
               }
          );
      }
}
```

Using a B-Tree

The implementation we've just gone through covers the data structure itself. By itself, the B-Tree class we've implemented could be used as a data storage container. To use the B-Tree as an index for a large data file, we need to take several additional steps.

In the beginning of the chapter, we learned that the B-Tree structure can be stored directly in a file. To do that, we first need to extend the BTree to handle the creation, reading, and writing of the index file.

Some methods we might use for these purposes are **`createIndexFile()`**, **`deleteIndexFile()`**, **`readMultiNode()`**, and **`writeMultiNode()`**. The **`createIndexFile()`** method most likely will take a string **`name`** as an argument. We then can use the name to create a file stream, which we can use as the disk-based index file. The **`deleteIndexFile()`** method, of course, will delete the entire index from the disk.

We can use the **`readMultiNode()`** and **`writeMultiNode()`** methods to read and write entire multi-way nodes to and from the tree structure. With these methods, we need to treat the binary representation of the multi-way node as though it were a more traditional implementation.

Page 235

Keep in mind that one of the big advantages of the B-Tree as an index to a data file is in reducing the number of reads and writes to disk. We accomplish this in traditional B-Trees by reading and writing nodes that are optimized in size to the disk device. If the binary representation is used and binary nodes are read and written to individually, we lose this advantage. Using the red-black notation enables us to easily reconstruct the nodes in a way that more closely resembles the traditional nodes.

Page 236

Exercises

1. Complete the implementation of the file-based B-Tree class by extending the B-Tree class and implementing the methods required to save and load the tree data.

2. Add a method to print the structure of the B-Tree both in its red-black form and as a traditional B-Tree structure with its multi-way nodes.

3. Create an ordered data set and compare the tree structures of simple binary trees, 2-3-4 trees, and B-Trees.

Page 237

Summary

In this chapter, we learned the following:

- A B-Tree is a variant of the tree structure that can have nodes containing an arbitrary number of keys and/or data structures.
- We looked at the functionality of the traditional B-Tree.
- We created a binary representation of the B-Tree similar to the red-black binary

implementation of the 2-3-4 tree.

- The B-Tree structure is commonly used to create and maintain index files for large data sets.

Appendix A
Java Language Overview

This appendix will present a brief overview of the Java programming language. This appendix is not intended as a Java primer but as a convenient and brief reference. The general syntax, keywords, primitive types, and class structure are briefly explained. An abbreviated overview of the Java core packages and class hierarchy is presented as well.

Java

In 1995, Sun Microsystems, Inc. released the first official version of the Java platform and programming language. Java is intended to be an operating system and hardware platform independent environment in which to develop and run computer programs. A special emphasis is placed on network-centric applications. Java achieves its platform independence by compiling the Java source code files into *byte-code* files that can be decoded by a Java runtime virtual machine. The virtual machine (VM) is a platform-specific byte code interpreter that translates the byte code into native instructions on the local host computer In this fashion, any platform that supports a Java Virtual Machine can run any Java application without the need to recompile or rewrite any of the source code.

Two basic execution units exist in the Java environment—Java applets and Java applications. *Applets* are executable units designed to run in a World Wide Web browser environment such as Netscape Navigator or Microsoft Internet Explorer. They are embedded in HTML encoded "Web Pages" and run in a protected environment in the context of the browser itself. The browser identifies an applet by recognizing a special HTML tag **`<APPLET>`** that supplies the browser with all of the information it needs in order to download and run the applet. Fields within the **`<APPLET>`** tag define where the byte code for the applet can be found as well as any parameters to be supplied to the applet at start-up. The browser then starts a Java VM and runs the byte code by making calls to specific methods defined by the **`java.applet.Applet`** class.

Java applications are designed to be run as stand-alone programs that do not need a Web browser or Web Pages. They still require a Java Virtual Machine to interpret the byte code Java class files, but they run in a

stand-alone Java runtime interpreter, which makes them roughly equivalent in execution context to native applications. The interpreter runs a Java application by calling the public static method main() in the class named on the interpreter's command line.

Security

Security issues must be taken into consideration when letting an unknown applet from an internet site run on a local machine. Allowing an unknown applet unrestricted access to the local host machine could lead to hostile attacks or viruses on the local computer running the applet. For this reason, security restrictions exist regarding what resources an applet can access.

Many World Wide Web browsers use a *sandbox* approach to applet security This approach defines a very limited environment in which all applets must run. The environment generally excludes any access to the following:

- The local files system (if any)
- Execution of any local commands
- Any socket connection except to the server from which the applet was loaded (its originating URL)
- Any GUI resources not managed by the browser itself (additionally, any windows opened by the browser on behalf of the applet get tagged with a warning such as **`Untrusted Applet Window`** to warn users that the window belongs to the applet)

Java applications generally have less restrictions than Java applets. A Java application does not run in the context of a web browser and so cannot generally be invoked directly over the internet. A Java application needs to be executed in an explicitly created Java Virtual Machine on the host computer unlike the applet, which is executed automatically by its browser host. This being the case, Java applications are considered to be as trusted as any other non-Java application and have the same access to system resources as any other application.

In either case, a Java **`SecurityManager`** class is used to determine the access characteristics of the Java executable unit being invoked. Although most web browsers come with a preconfigured SecurityManager, the SecurityManager object may be modified to allow different access rights

based upon criteria such as the originating URL or a digital certificate or signature attached to the apple.

Keywords

The following words are reserved in the Java language:

- abstract
- boolean
- break
- byte

- case
- catch
- class
- const[2]
- continue
- do
- default
- double
- else
- extends
- false[1]
- final
- finally
- float
- for
- goto[2]
- if
- int
- interface
- long
- native
- new
- null[1]
- package
- private
- protected
- public
- return

- short
- static
- super
- switch
- synchronized
- this
- throw
- transient
- true[1]
- try

---

[1]True, false, and null are defined in the language as literal values. Unlike the rest of the list, they are not keywords. They are reserved though, in that they cannot be used as identifiers.

[2]These words are reserved but currently unused in the language.

- implements
- import
- instanceof
- void
- volatile
- while

None of the words listed may be used as identifiers in any Java language construct. All of these words have specific meanings in the Java language and will generate compiler errors if not used appropriately.

Java Built-In Data Types

Two basic sets of data types exist in the Java language. The first set is known as the *primitive types*. The second set is the *reference types.*

Primitive Types

The Java primitive types are the same basic data types found in almost any programming language. Primitive types hold a single data item. They do not have any methods associated with them and are not considered as objects in Java. Some of the reserved words in Java

represent these primitive types.

The defined integer types are byte, short, int, and long. All integer types in Java are signed and so one bit of the total number of bits in each type is used for the sign. The floating point types are float and double. Java floating point types are IEEE 754-1985 compliant. The Java char type

TABLE A.1
Primitive Types

Keyword	Represents Values
byte	8-bit signed integer
short	16-bit signed integer
int	32-bit signed integer
long	64-bit signed integer
float	32-bit floating point number
double	64-bit floating point number
char	16-bit unicode character
boolean	true or false

holds a 16-bit Unicode 1.1 character value, and the boolean type holds either the true or false literal.

In other programming languages such as C and C++, the size of the primitive types is dependent upon the platform on which the program is compiled/run. For example, in C++ an integer (int) may be defined to be 16 bits on a PC platform and 32 bits on a UNIX platform. One of the advantages that Java has over these other languages is the fact that the language itself defines the size of the primitive types. This property of the language removes many of the cross-platform porting issues associated with these other languages.

Reference Types

The second set of data types in Java are the reference types. Like most object-oriented languages, the Java language uses classes as the constructs that define objects. Any field or variable associated with a class or an interface is a reference type. Arrays are also considered to be reference types regardless of the contents of the array. So an array primitive types is still a reference object.

A variable associated with a reference type is called a *reference*. A reference to an object is basically a handle or pointer to that object. Under the Java platform, there is never direct access to the memory that holds an object. All access to objects is through one of these handles (or references).

Access Modifiers

The Java language has four levels of access: private, default (or package), protected, and public. Three Java keywords cover these four levels. The default access in Java is defined to be the package access. Package access is assumed unless one of the specific keyword modifiers are used to define the access level. The access levels are hierarchical in relation to

one another

The private modifier is the most restrictive as it limits access to the enclosing scope of the declaration. The private modifier may be used to restrict the access of any member of a class (field or method) when it is defined. Any field declared as private in a class is considered to be tightly encapsulated because it is inaccessible to any class except its own. Any method in the defining class can freely access such a field.

Generally, if a private field is used to describe an attribute of the class that needs to be set or read publicly, access or methods are provided. In

Figure A-1
Access levels defined in a simple class.

```
Class MyClass
{
     public void setState( boolean b )
     {
          state = b;
     }

     public boolean getState()
     {
          return state;
     }

     private boolean state;
}
```

Figure A-1, **`MyClass`** has one private member field, the boolean *state*. Since *state* is private, it can be accessed only by a member method in the class **`MyClass`**, such as setState() and getState(). An external user of the class gains access through these two methods thereby insulating the actual data field from outside access.

The default access (also known as package access) expands the access level to include the entire package. If no modifier is applied to a field or method member of a class, it is considered to be of package access. Package level access restricts any class outside of the declared package from accessing the member field or method. Inversely, this means that any class belonging to the same package has unrestricted access to the member.

As with the private modifier, a field with package access may describe an attribute that may need to be publicly read or set. If this is the case, the class **`API`** will include accessor methods for the field. C++ developers will find that package access is similar in concept to the friend keyword in C++.

The protected modifier, in turn, expands this access to include subclasses, even those defined in other packages. Protected access still includes all the access granted by the package level. With the protected modifier, subclasses can access member fields and methods from within the superclass. Protected access is often given for methods that would otherwise be restricted to private access. The difference is that the superclass designer assumes that the method may want to be changed for a subclass. This allows subclasses to modify or extend the behavior of a

class without the need to declare either method or field members as publicly accessible.

The public modifier removes all restrictions on access, which gives the Java equivalent of global access. Any method from any class can access members declared as public. The public modifier is generally reserved for member methods because public access to a member field is generally considered to be a bad idea since it breaks encapsulation.

Packages

The Java language allows the grouping of related classes using a concept called packages. Classes and their members that belong to the same package have special privileges in regard to one another as explained earlier in this appendix.

The package for a class is declared as the first non-commented statement in a source file. The package name becomes part of the fully qualified name of the class. For example, **`java.util.Vector`** is the fully qualified name of the Vector class that belongs to the java.util package. In turn the **`java.util`** package is nested within the java package. Package names are separated from nested or sub-packages by using the "." notation, as is the name of the package from the name of the class.

To provide an easier way to refer to classes and thereby make code a bit more readable, Java allows a source file to declare that specific classes or packages are to be used in the source. The import statement provides this functionality:

```
import java.util.*;
import java.io.File;
```

These two declarations denote that all of the **`java.util`** package and the File class of the **`java.io`** package may be referenced in the enclosing source file without the need for a fully qualified class name. The "**`*`**" notation indicates that the entire package is being imported as opposed to naming each individual class. In this example, **`Vector`** may be used in place of **`java.util.Vector`** anywhere in the source file. **`File`** will also refer to **`java.io.File`** throughout the source file.

Classes

The class keyword is used to denote a class definition:

```
class MyClass {...}
```

A class name declaration is always followed by a definition block, which is delimited by a matching set of curly braces "{ }". The definition block encloses all of the member fields and methods contained in the class.

A class may be declared as allowing public or package access to itself. A public class is accessible from anywhere while a package (or default) class

is accessible only from within the package of which it is declared to be a member.

A class may also be declared as abstract or final. An abstract class may not be directly instantiated. It must be subclassed, and the subclass may be instantiated. The abstract class may be used as the type of a reference variable and thereby hold any object that is instantiated as a subclass of the abstract class. A final class may be instantiated directly but may not be extended (subclassed).

Classes may be extended to provide additional specialization or functionality. A class that extends another class is called a *subclass* The class being extended is called a *superclass* The Java language limits the number of superclasses a subclass can extend to one. In other words, multiple inheritance is not a supported feature in the Java language.

```
class MyClass
     extends SomeOtherClass
{
. . .
}
```

Subclasses automatically inherit all of the public and protected members of the declared superclass. This includes the methods and the fields defined by the superclass. Private members are not accessible to the subclass and package members are available only if the subclass belongs to the same package as the superclass.

Interfaces

The Java language allows for a construct called an interface. An *interface* is a definition of a named API with no associated implementation. The API definition may contain zero or more method declarations and zero or more static field declarations.

```
public interface MyInterface
{
     public void iMethod( Sting s );
}
```

A class may be defined to conform to an interface by declaring that the class *implements* the interface and by providing implementations for the

declared methods. The fields defined by the interface are automatically inherited by any class that implements the interface.

```
public class MyClass
      implements MyInterface
{
      public void iMethod( String s )
      {
           . . .
      }

      . . .
{
```

A class may implement as many interfaces as is desired. This faculty to implement multiple

interfaces offers developers the ability to design type-safe APIs that can be used to bridge between classes.

Methods

Methods are the routines that perform the work of classes (objects). Each class can define its own methods to perform any task for which the class is responsible.

Every method has a *signature.* A method signature defines the name of the method and the number and types of the arguments. All arguments are passed to the method by value and are strictly type checked. Each method also has a return type that may be void, one of the primitive types, or a class. A value of the return type (except for void, which returns nothing) is passed back to the calling method upon completion of the method.

Public or protected methods inherited from a superclass may be overridden by a subclass. An overriding method in a subclass has the same signature as the method that is being overridden in the superclass. The overriding method must also have the same return type as the overridden method.

A subclass may access an overridden method in its superclass by using the super keyword:

```
super.methodName( args );
```

Otherwise a call to the overridden method from the subclass will result in a call to the overriding method in the subclass.

Applications and Applets

The two basic execution units in Java are applications and applets. Applets run inside World Wide Web browsers. All Java applets are subclasses of the **`java.applet.Applet`** class. Special HTML **`<APPLET>`** tags are embedded in web pages that give the browser the information needed to load and start the applet. When an applet is loaded into the browser's Java Virtual Machine, specific methods in the applet superclass are invoked to initialize and start the applet.

Applications run as stand-alone programs. A Java Virtual Machine is started as a process on the native operating system which in turn loads the desired class and invokes the class's static main method.

The Java Core Class Library

The Java Development Kit (JDK) from JavaSoft[3] comes with a set of core classes defined by JavaSoft to provide the minimum level of functionality available to the Java platform. The core library is divided into related packages based on the types of functionality the included classes provide. The following sections will give a summary explanation of each package in the core library including a brief overview of the specific classes in the package.

The `java.applet` Package

The **java.applet** package is the basis for all applets. Any Java process to be run from a browser platform must extend the Applet class. The Applet class itself extends **java.awt.Panel** and so provides a GUI container class for all of the applet's interface components.

---

[3]JavaSoft is a business unit of Sun Microsystems, Inc. JavaSoft maintains and develops the Java platform and related Java technologies.

Interfaces

AppletContext
AppletStub
AudioClip

Classes

Applet

The **java.Applet.Applet** class is derived from the **java.awt.Panel** class. This derivation gives each Applet a base GUI container in which to run.

The java.awt Package

java.awt is the abstract windowing toolkit package. This package is the basis for the graphical user interfaces (GUI's) developed in Java. **java.awt** supplies standard GUI components such as buttons, lists, labels, panels, and canvases that can be used directly or subclassed into custom components:

Interfaces

Adjustable
ItemSelectable
LayoutManager
LayoutManager2
MenuContainer
PrintGraphics
Shape

Classes

AWTEvent
AWTEventMulticaster
BorderLayout
Button
Canvas

CardLayout
Checkbox
CheckboxGroup

```
CheckboxMenuItem
Choice
Color
Component
Container
Cursor
Dialog
Dimension
Event
EventQueue
FileDialog
FlowLayout
Font
FontMetrics
Frame
Graphics
GridBagConstraints
GridBagLayout
GridLayout
Image
Insets
Label
List
MediaTracker
Menu
MenuBar
MenuComponent
MenuItem
MenuShortcut
Panel
Point
Polygon
PopupMenu
PrintJob
Rectangle
ScrollPane
Scrollbar
SystemColor
TextArea
TextComponent
TextField
Toolkit
Window
```

Exceptions

```
AWTException
IllegalComponentStateException
```


Errors

```
AWTError
```

The `java.awt.datatransfer` Package

The **`java.awt.datatransfer`** package provides an interface to the native concept of a clip board, allowing Java classes to perform cut-and-paste operations to the native clip board. This allows a Java application to exchange data with another application running in the same native environment.

Interfaces

`ClipboardOwner`
`Transferable`

Classes

`Clipboard`
`DataFlavor`
`StringSelection`

Exceptions

`UnsupportedFlavorxception`

The `java.awt.event` Package

The **`java.awt.event`** package contains classes related to the delegation event model introduced in JDK Version 1.1. Besides the event classes themselves, this package contains the interfaces necessary to implement listener functionality as well as default adapter classes which implement no-op listeners that may easily be extended.

Interfaces

`ActionListener`
`AdjustmentListener`
`ComponentListener`
`ContainerListener`
`FocusListener`
`ItemListener`
`KeyListener`
`MouseListener`
`MouseMotionListener`
`TextListener`
`WindowListener`

Classes

`ActionEvent`
`AdjustmentEvent`
`ComponentAdapter`
`ComponentEvent`
`ContainerAdapter`
`ContainerEvent`
`FocusAdapter`
`FocusEvent`
`InputEvent`
`ItemEvent`

KeyAdapter
KeyEvent
MouseAdapter
MouseEvent
MouseMotionAdapter
PaintEvent
TextEvent
WindowAdapter
WindowEvent

The `java.awt.image` Package

The **`java.awt.image`** package contains several classes that are useful to manipulate images:

ImageConsumer
ImageObserver
ImageProducer
AreaAveragingScaleFilter
ColorModel

CropImageFilter
DirectColorModel
FilteredImageSource
ImageFilter
IndexColorModel
MemoryImageSource
PixelGrabber
RGBImageFilter
ReplicateScaleFilter

The `java.io` Package

The **`java.io`** package provides classes that handle all of the basic input and output for a Java process. This package includes classes that handle raw and streamed input and output:

Interfaces

DataInput
DataOutput
Externalizable
FilenameFilter
ObjectInput
ObjectInputValidation
ObjectOutput
Serializable

Classes

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
ByteArrayInputStream

ByteArrayOutputStream
ByteToCharConverter
CharArrayReader
CharArrayWriter
CharToByteConverter
DataInputStream
DataOutputStream
File
FileDescriptor
FileInputStream

FileOutputStream
FileReader
FileWriter
FilterInputStream
FilterOutputStream
FilterReader
FilterWriter
InputStream
InputStreamReader
LineNumberInputStream
LineNumberReader
ObjectInputStream
ObjectOutputStream
ObjectStreamClass
OutputStream
OutputStreamWriter
PipedInputStream
PipedOutputStream
PipedReader
PipedWriter
PrintStream
PrintWriter
PushbackInputStream
PushbackReader
RandomAccessFile
Reader
SequenceInputStream
StreamTokenizer
StringBufferInputStream
StringReader StringWriter
Writer

Exceptions

CharConversionException
ConversionBufferFullException
EOFException
FileNotFoundException
IOException
InterruptedIOException
InvalidClassException
InvalidObjectException
MalformedInputException
NotActiveException

NotSerializableException
ObjectStreamException
OptionalDataException
StreamCorruptedException
SyncFailedException
UTFDataFormatException

UnknownCharacterException
UnsupportedEncodingException
WriteAbortedException

The `java.lang` Package

The **`java.lang`** package is the base package for the Java Development Kit. It is a globally available set of classes that require neither an import statement nor full package qualification in order to be used. The classes in this package include all of the classes used to instantiate a process and the threads that run in that process, the wrapper classes for the primitive types, the basic numeric and text handling classes, and the Object class which is the ultimate superclass of all Java classes.

Interfaces

Cloneable
Runnable

Classes

Bignum
Boolean
Byte
Character
Class
ClassLoader
Compiler
Double
Float
Integer
Long
Math
Number
Object
Process
Runtime
SecurityManager
Short
String
StringBuffer
System
Thread

ThreadGroup

```
Throwable
Void
```

Exception Classes

```
ArithmeticException
ArrayIndexOutOfBoundsException
ArrayStoreException
ClassCastException
ClassNotFoundException
CloneNotSupportedException
Exception
IllegalAccessException
IllegalArgumentException
IllegalMonitorStateException
IllegalStateException
IllegalThreadStateException
IndexOutOfBoundsException
InstantiationException
InterruptedException
NegativeArraySizeException
NoSuchFieldException
NoSuchMethodException
NullPointerException
NumberFormatException
RuntimeExcept ion
SecurityException
StringIndexOutOfBoundsException
```

Errors

```
AbstractMethodError
ClassCircularityError
ClassFormatError
Error
ExceptionInInitializerError
IllegalAccessError
IncompatibleClassChangeError
InstantiationError
InternalError
LinkageError
NoClassDefFoundError
NoSuchFieldError
NoSuchMethodError
OutOfMemoryError
StackOverflowError
ThreadDeath
UnknownError
```



```
UnsatisfiedLinkError
VerifyError
VirtualMachineError
```

The `java.lang.reflect` Package

The **`java.lang.reflect`** package provides the means to interrogate classes at run time as to the public data and methods provided by the class. The facility is used extensively in Java Beans.

Interfaces

`Member`

Classes

`Array`
`Constructor`
`Field`
`Method`
`Modifier`

Exception Classes

`InvocationTargetException`

The `java.net` Package

The **`java.net`** package provides access to network resources. This includes Internet and local area network resources. TCP/IP sockets and URL connections are among the facilities offered by this package.

Interfaces

`ContentHandlerFactory`
`FileNameMap`

`SocketImplFactory`
`URLStreamHandlerFactory`

Classes

`ContentHandler`
`DatagramPacket`
`DatagramSocket`
`DatagramSocketImpl`
`HttpURLConnection`
`InetAddress`
`MulticastSocket`
`ServerSocket`
`Socket`
`SocketImpl`
`URL`
`URLConnection`
`URLEncoder`
`URLStreamHandler`

Exception Classes

indException
ConnectException
MalformedURLException
NoRouteToHostException
ProtocolException
SocketException
UnknownHostException
UnknownServiceException

The `java.rmi` Package

The **`java.rmi`** package provides remote method invocation. This allows a Java process running on one virtual machine to invoke methods on an object running on another virtual machine.

Interfaces

Remote

Classes

Naming
RMISecurityManager

Exception Classes

AccessException
AlreadyBoundException
ConnectException
ConnectIOException
MarshalException
NoSuchObjectException
NotBoundException
RMISecurityException
RemoteException
ServerError
ServerException
ServerRuntimeException
StubNotFoundException
UnexpectedException
UnknownHostException
UnmarshalException

The `java.rmi.dgc` Package

The **`java.rmi.dgc`** package provides distributed garbage collection for remote objects.

Interfaces

DGC

Classes

Lease
VMID

The `java.rmi.registry` Package

The **`java.rmi.registry`** package provides the classes used in the RMI registry. Any virtual machine allowing rmi connections will have a registry. A registry on a given node provides a database that maps local names to remote objects.

Interfaces

Registry
RegistryHandler

Classes

LocateRegistry

The `java.rmi.server` Package

The **`java.rmi.server`** package provides the server-side functionality for RMI services. The server is the node on which RMI connections and requests are received and processed.

Interfaces

LoaderHandler
RMIFailureHandler
RemoteCall
RemoteRef
ServerRef
Skeleton
Unreferenced

Classes

LogStream
ObjID
Operation
RMIClassLoader

RMISocketFactory
RemoteObject
RemoteServer
RemoteStub
UID
UnicastRemoteObject

Exception Classes

ExportException

ServerCloneException
ServerNotActiveException
SkeletonMismatchException
SkeletonNotFoundException
SocketSecurityException

The `java.security` Package

The **`java.security`** package provides the means to examine a class at load-time to determine the classes' level of access to the local system.

Interfaces

Certificate
Key
Principal
PrivateKey
PublicKey

Classes

DigestInputStream
DigestOutputStream
Identity
IdentityScope
KeyPair
KeyPairGenerator
MessageDigest
Provider
SecureRandom
Security
Signature
Signer

Exception Classes

DigestException
InvalidKeyException
InvalidParameterException
KeyException
KeyManagementException
NoSuchAlgorithmException
NoSuchProviderException
ProviderException
SignatureException

The `java.security.acl` Package

The **`java.security.acl`** package provides an access control list functionality to the java.security package.

Interfaces

Acl

AclEntry
Group
Owner
Permission

Exception Classes

AclNotFoundException
LastOwnerException
NotOwnerException

The `java.security.interfaces` Package

The **`java.security.interfaces`** package defines the interfaces necessary to implement encryption schemes in Java.

Page 263

Interfaces

DSAKey
DSAKeyPairGenerator
DSAParams
DSAPrivateKey
DSAPublicKey

The `java.sql` Package

The **`java.sql`** package provides an interface to the industry standard Structured Query Language used in accessing relational databases. The classes in this package use and manage JDBC drivers that allow Java to interface with native RDBMS's.

Interfaces

CallableStatement
Connection
DatabaseMetaData
Driver
PreparedStatement
ResultSet
ResultSetMetaData
Statement

Classes

Date
DriverManager
DriverPropertyInfo
Time
Timestamp
Types

Exception Classes

DataTruncation
SQLException

`SQLWarning`

The `java.text` Package

The classes in the **`java.text`** package provide a language-independent way to handle the processing and formatting of text.

Interfaces

`CharacterIterator`

Classes

`BreakIterator`
`ChoiceFormat`
`CollationElementIterator`
`CollationKey`
`Collator`
`DateFormat`
`DateFormatSymbols`
`DecimalFormat`
`DecimalFormatSymbols`
`FieldPosition`
`Format`
`MessageFormat`
`NumberFormat`
`ParsePosition`
`RuleBasedCollator`
`SimpleDateFormat`
`StringCharacterIterator`

Exception Classes

`ParseException`

The `java.util` Package

The **`java.util`** package contains various utility classes. The package includes classes that handle time and date functionality. There are container classes like the Vector, Hashtable, Dictionary, and Stack. The package also includes the Random class used to generate random numbers. The **`java.util`** package is also the home of the Enumeration and Observer interfaces.

Interfaces

`Enumeration`
`EventListener`
`Observer`

Classes

BitSet
Calendar
Date
Dictionary
EventObject
GregorianCalendar
Hashtable
ListResourceBundle
Locale
Observable
Properties
PropertyResourceBundle
Random
ResourceBundle
SimpleTimeZone
Stack
StringTokenizer
TimeZone
Vector

Exceptions

EmptyStackException
MissingResourceException
NoSuchElementException
TooManyListenersException

The `java.util.zip` Package

The **`java.util.zip`** package provides the classes needed for data compression and decompression using various algorithms.

Interfaces

Checksum

Classes

Adler32
CRC32
CheckedInputStream
CheckedOutputStream
Deflater
DeflaterOutputStream
GZIPInputStream
GZIPOutputStream
Inflater
InflaterInputStream
ZipEntry
ZipFile
ZipInputStream
ZipOutputStream

Exception Classes

```
DataFormatException
ZipException
```


Appendix B
Keywords and Literals

The following keywords and literals are defined for the Java language:

abstract
A keyword used to identify a class or method purposely left without a complete implementation with the expectation that a subclass will provide the implementation.

boolean
A primitive type which always holds a value of either true or false.

break
A keyword used to leave an execution loop or switch statement.

```
while( true )
{
     if( state == true )
          break;
{
```

byte
A primitive integer type holding an 8-bit value.

case
A control flow statement used in a switch construct. Defines an action block to be executed if the switch constant and the case constant are equal. See switch.

catch
A keyword used in exception handling to indicate the type of exception being trapped.

```
try
{
. . .
}
catch( IOException e )
{
. . .
}
```


class
A keyword used to indicate the beginning of a class definition.

const
A word reserved for future use in the Java language.

continue
A keyword used to switch program flow to the next iteration of a loop.

```
while( i < 100 )
{
     if( i % 10)
          continue;
     else
          . . .
}
```

do
A keyword used to indicate the starting point of a loop that has its exit contition after the body of the loop. Always used with the while keyword.

```
do
{
. . .
} while( b == true );
```

default
A keyword indicating the default action in a switch statement if none of the `case` statements apply. See switch.

double
A primitive floating point numeric type that holds a 64-bit value.

else
A keyword indicating the alternative branch in an if-else conditional statement.

```
if( b == true
   . . .
else
   . . .
```

extends
A keyword used in a subclass definition statement defining the name of the superclass from which this class derives.

false
A boolean literal.

final
A keyword used to identify a method that is not overridable or a field that is not modifiable.

finally
A keyword used in the definition of a try/catch/finally block used for exception handling. The finally block is always executed regardless of whether or not the exception condition occured.

float
A primitive floating point numeric type that holds a 32-bit value.

for
A keyword used in the definition of an iteractive loop.

```
For( int i = 0; i < 10; i++  )
{
   . . .
}
```

if
A keyword used in the definition of a conditional statement. If the condition resolves to true, the supplied statement/block is executed.

```
if( b == true )
     . . .
```

implements
A keyword used with a class declaration to identify the interfaces implemented by the class.

import
A keyword used to identify the classes/packages from which the source file will use shortened class and interface names.

`import java.awt.*;` allows `"Graphics"` instead of the fully qualified class name `"java.awt.Graphics"`

instanceof
A keyword used to test a reference variable to determine whether the variable is castable to the supplied type.

```
if( b instanceof MyClass )
     . . .
```

int
A primitive integer type holding a 32-bit value.

interface
A keyword used to declare a Java interface.

long
A primitive integer type holding a 64-bit value.

native
A keyword used to denote that a method is defined in native (non-portable) code.

new
A keyword used to allocate dynamic memory for a class or array instance.

null

A literal used to indicate that a reference does not point to any object.

package
A keyword used to define the package to which a source file belongs. The package statement must be the first statement in the source file.

private
A keyword access modifier used to denote access limited to the defining scope.

protected
A keyword access modifier used to denote access limited to the defining package and any subclasses of the defining class.

public
A keyword access modifier used to denote free access by any class or method regardless of package or class hierarchy.

return
A keyword used in a method definition to indicate the exit point of the method. It may also be used to supply the defined return value.

short
A primitive integer type holding a 16-bit value.

static
A keyword used to denote that a member exists only once per class rather than once per instance of the class.

super
A keyword used to refer to the superclass of this instance.

switch
A keyword used in the definition of a multi-conditional statement. A supplied value is tested against multiple "cases" or constants for equivalence. Execution is transferred to the appropriate case.

synchronized
A keyword used to place a lock and monitior on an object in order to ensure data integrety in objects accessed by multiple threads.

this
A keyword referring to the current instance of a class. The current object.

throw
A keyword used to generate and deliver an exception.

throws
A keyword used in the declaration of a method to indicate the possible exception conditions that may be thrown by the method.

transient
A keyword used to denote that a field in a class object will not be persistent through serialization.

true
A boolean literal.

try
A keyword used to define a block for which specific exceptions should be caught. See catch.

void
A keyword used to indicate that a method has no return value.

volatile
A keyword used on a data field to indicate that optimizing compilers should not make assumptions about the field.

while
A keyword used to define a conditional loop.

```
While( b == true )
{
. . .
}
```


What's on the CD-ROM?

Complete Source Code for Abstract Data Types in Java

All of the source code listed in the book is provided, ready to compile. The source listings are broken down chapter-by-chapter Each chapter's Java source comes with an example **`makefile`** that can be used to easily build all of the classes described in the book.

The main directory for all of the source code is \adt on the CD-ROM. Each chapter has its own subdirectory containing all of the source for that chapter.

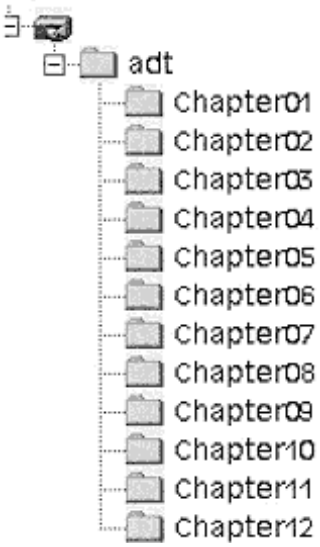


Figure 1
The main directory for the source
code on the CD-ROM.

Java Development Kit (JDK) Version 1.1.3

The complete 1.1.3 release of the Java Development Kit Version 1.1.3 is included on the CD-ROM. The compressed, installable versions for Windows 95, Windows NT, Solaris 2.x (Sparc), and Solaris 2.x (x86) are located in the JDK directory. The file names follow.

`jdkl.l.3-beta-solaris2-x86.bin`
Solaris 2.x PC (x86)
version (beta version)

`jdk1.1.3-solaris2-spare.bin`
Solaris 2.x Sparc
version

`jdk113.exe`
Windows 95 / Windows NT
version

`jdk113doc.tar.gz`
Complete documentation for the JDK
in a gzipped tar file format.

Installing the JDK on Windows 95 / Windows NT

To install the JDK on Windows 95 or Windows NT, insert the CD-ROM, select the appropriate drive letter in Windows explorer or File Manager, and double-click on the **`jdk113.exe`** file. The installation program will walk you through all of the steps necessary to complete installation.

Please read the **`\jdk\README`** file in order to set up the appropriate environment variables

before attempting to run the JDK utilities.

Installing the JDK on Solaris

To install the JDK on either Solaris platform, Sparc, or x86, copy the appropriate file from the CD-ROM to the desired base directory on your workstation. If you unpack the software or documentation in a directory that contains a directory named **`jdk1.1.3`,** the new software will overwrite files of the same name in that **`jdk1.1.3`** directory. Please be careful to rename the old directory if it contains files you want to keep.

In a shell window, execute the following commands. Note that executing these commands temporarily creates a README file in the current directory (which will overwrite any README file you may have).

To install the JDK on the SPARC platform (must be unpacked on a SPARC machine running Solaris 2.4 or greater):

```
% chmod a+x jdk1.1.3-solaris2-sparc.bin
% ./jdkl.1.3-solaris2-sparc.bin
```

To install the JDK on the x86 platform (must be unpacked on a x86 machine running Solaris 2.5 or greater):

```
% chmod a+x jdk1.1.3-beta-solaris2-x86.bin
% ./jdk1.1.3-beta-solaris2-x86.bin
```


This will bring up a license for you to read. If you agree, type **`yes`**, press Return, and the program creates a directory called **`jdk1.1.3`** containing the JDK software.

Included in the unpacked files is a file **`lib/classes.zip`. Do not unZIP the `classes.zip file`.** This file contains all of the core class binaries and must remain in its zipped form for the JDK to use it.

The Java source files originally appear as a **`src.zip`** file under the **`jdk1.1.3`** directory in the Solaris installation, which you may unzip manually to obtain access to the source code for the JDK class libraries. However, you must use an unzip program that maintains long filenames. Such unzip utilities may be found at the UUNet FTP Site (**`ftp://ftp.uu.net/pub/archiving/zip`**)

Please read the **`\jdk\README`** file to set up the appropriate environment variables before attempting to run the JDK utilities.

Installing the JDK Documentation

To install the javadoc (HTML) documentation for all of the Java classes, copy the **`jdk113doc.tar.gz`** file into the same directory into which you installed the JDK. Uncompress the file. On Windows platforms, you need to have the WinZip utility **(`http://www.winzip.com`).** On the Solaris platforms, you need the gzip utility (available from countless places on the Internet). The compressed file expands into the **`jdkl.1.3/docs`** directory that will contain all of the javadoc files for the core libraries.

Running the JDK from the CD-ROM

If you are running Windows 95 or Windows NT, you can run the JDK directly from the CD-ROM without having to install it on your hard drive. The **\jdkl.1.3** directory on the CD-ROM contains a completely expanded and ready-to-use version of the JDK, including the Java source files. You need to read the **\jdk\README** file to set up the appropriate environment variables.

About ObjectSpace

ObjectSpace is an advanced software technology company that specializes in distributed computing. Using a partnership approach, ObjectSpace provides the people, process, technology, and skills transfer services needed to render innovative business solutions for our clients. ObjectSpace strives for the highest levels of quality and functionality while transforming the client's development organization.

Extensive experience and continued research make the people at ObjectSpace experts at the application of distributed technology. We facilitate skills transfer through an educational services curriculum that blends partnered project development, training classes, and mentoring services. ObjectSpace leverages its rapid application development process by relying on proven, iterative methodology. Finally, the company's portfolio of application frameworks, advanced technology components, and project management tools ensure visible, timely, and measurable results.

Headquartered in Dallas, Texas, ObjectSpace maintains offices in major cities across the US. and has a worldwide distributor network. Among our clients are Fortune 500 companies specializing in manufacturing, communications, and financial services.

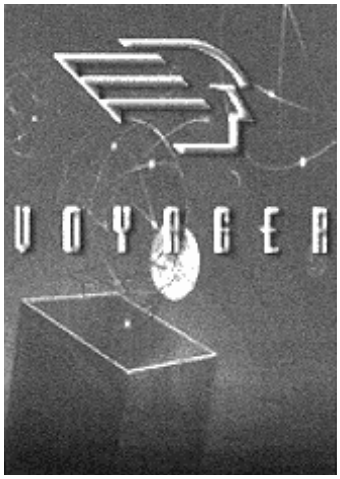
The ObjectSpace Development Technology Division leads the market for standards-driven C++ and JavaTM components. ObjectSpace products are proven performers that deliver advanced technology in a form easily applied to today's development problems.

ObjectSpace Java™ Products

The Java Products from ObjectSpace provide Java developers with core technology for Java development. Both are *free for commercial use at www.objectspace.com.*

- JGL™—Generic Collection Library for Java. The most comprehensive set of containers and algorithms available for Java today.

- Voyager™—Agent-Enhanced Distributed Computing for Java. The only Java platform today that seamlessly supports traditional and agent-enhanced distributed computing techniques



What is Voyager?

Voyager is the world's first 100-percent Java agent-enhanced Object Request Broker (ORB). Voyager enables Java programmers to create sophisticated network applications quickly and easily using bot traditional and agent-enhanced distributed programming techniques.

Voyager was designed carefully to be extremly easy to use. Use regular Java message syntax to construct remote objects, send them messages, and move them between applications. In minutes, you can create autonomous agents that can roam a network and continue to execute as they move.

Traditional Distributed Computing

Many people today are becoming familiar with client/server programming. Well-known technologies, such as remote process communication (RPC), were designed specifically for the client/server paradigm. In client/server computing, the client establishes a connection with one or more stationary servers and sends them messages to complete a task.

Client/Server techniques do the following:

- Consume network bandwidth for each message.

- Require the network connection be maintained with a specific service during the entire conversation.

Agent-Based Computing

With agent-based computing, an application is constructed from a mix of stationary objects and mobile objects, or agents. When necessary, agents can move to stationary objects or to other agents to perform high-speed, local communications.

- Consumes network bandwidth only once, when the agent moves.
- Agents continue to execute after they move, even if they lose connectivity with their creators.
- Agents use high-speed native messaging to complete the conversation, consuming no network bandwidth.

The Best of Both Worlds

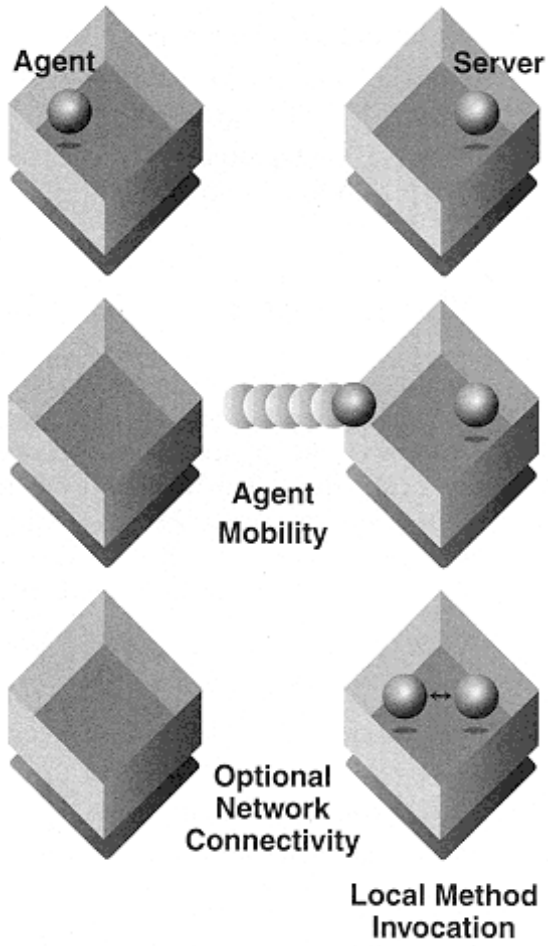
Voyager is the only Java platform today that seamlessly supports traditional and agent-enhanced distributed programming techniques. Depending on the application context, one of these two approaches may better satisfy the system requirements. In most large distributed systems, both techniques used together produce the optimal result. Don't restrict yourself by choosing only one-choose Voyager and get both. Voyager is the obvious choice for distributed system development because it is

- 100% Java
- Extremely easy to use, requiring no modifications to your classes.
- A single unified platform providing remote messaging, mobility, and autonomy.
- Very compact and fast.
- Absolutely free for commercial use.

"Agent technology will be as important for the Internet as the Internet has been for personal computing. Voyager is the most powerful and easy-to-use solution for agent-enabled distributed computing I have seen."
—John Nordstrom, Sabre Decision Technologies

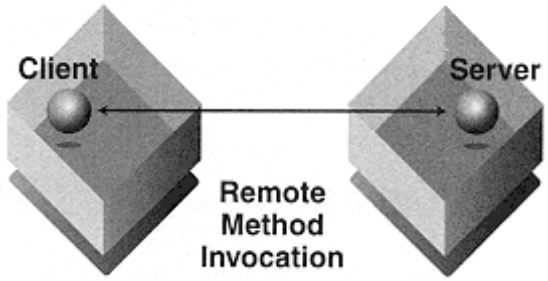
Transparently locate agents and send them messages as they work, even if the agents are moving. Do all this and much more—absolutely free—with Voyager. With Voyager you get the following:

VOYAGER INCLUDES SEAMLESS SUPPORT FOR AGENT TECHNOLOGY. Voyager enables you to create—in minutes—agents that can roam a network and continue to execute as they move. Because an agent is just a special kind of object, moving agents and other objects can exchange remote messages using regular Java message syntax.



VOYAGER DOES NOT REQUIRE YOU TO ALTER JAVA CLASSES IN ANY WAY. Voyager can remotely construct and communicate with any Java class, even third-party libraries without source. Other

technologies typically require the use of `.idl` files, interface definitions, and modifications to the original class, which consume development time and tightly couple your domain classes to a particular ORB technology.



VOYAGER HAS THE BEST INTEGRATION WITH THE JAVA LANGUAGE. Objects can be constructed remotely using the regular Java construction syntax; static methods can be executed remotely; remote exceptions are automatically rethrown to the caller; and serializable objects can be passed and returned by value. All of these features result in simpler and quicker

development for a Java programmer

VOYAGER INCLUDES SEAMLESS SUPPORT FOR OBJECT MOBILITY. You may move any serializable object after it has been created to a new location, even while the object is receiving messages. Messages sent to the old location are automatically forwarded to the new location.

VOYAGER IS FAST. Remote messages are as fast as the CORBA ORBS. In addition, messages delivered by mobile agents are often up to 100,000 times faster than other Java ORBs.

VOYAGER IS SMALL. The total class file size for Voyager is less than 150K. It is a fully functional agent-enhanced object request broker and does not require any additional software beyond the JDK 1.1.

VOYAGER IS COMPREHENSIVE. Version 1.0 includes support for one-way, future, and sync messages using TCP communications. Future versions will include a powerful distributed event system, group communications, a distributed directory service, store-and-forward, reliable UDP communications, mobile tracking facilities, and enhanced agent capabilities.

VOYAGER HAS THE BEST CONNECTIVITY. Some ORBs prevent objects in browsers from communicating with objects that are not located in the browser's server. Voyager does not impose this restriction and includes an integrated software router than enables objects to communicate with each other anywhere in the world (firewalls permitting).

VOYAGER IS 100% PURE JAVA. Voyager applications can be written once and run anywhere.

Voyager on the CD-ROM

The current beta 3.0 version of Voyager in included on the CD-ROM. All of the files needed to install Voyager are included. Please read the license agreement included with the software. By installing or using this software, you indicate acceptance of the terms and conditions of the enclosed license. You do not need to download additional materials to install beta 3.0. Updates are available for free directly from the ObjectSpace web site at the following address:

```
http://www.objectspace.com
```

The files on the CDROM are as follows:

`\Voyager\README.txt`
Installation instructions—PLEASE READ FIRST

`\Voyager\license.txt`
Voyager License—READ BEFORE INSTALLATION

`\Voyager\Voyager.PDF`
Voyager User's Manual in Adobe Acrobat format

`\Voyager\voyager_l.0_beta_3.0.zip`

Windows ZIP format

`\Voyager\voyager_l.0_beta_3.0.tar.Z`
UNIX TAR file

`\Voyager\voyager_l.0_beta_3.0.tar.gz`
UNIX GZipped TAR file

`\Voyager\voyager_l.0_beta_3.0.exe`
Self-extracting executable

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

S

T

U

V

W

About the Author

Michael S. Jenkins is an experienced object-oriented software development consultant who has been working with Java since its pre-Alpha release. He helped create the Chicago Board of Trade's "100% Java" electronic trading system. Mike has also developed successful system solutions for the Chicago Stock Exchange, Baxter Healthcare, and A.C. Nielsen/Dun & Bradstreet.