O'REILLY®

Java: The Legend

Past, Present, and Future

Ben Evans

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

Java: The Legend

Past, Present, and Future

Ben Evans

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Java: The Legend
by Ben Evans

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Nan Barber
Production Editor: Nicholas Adams
Proofreader: Nicholas Adams
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Anna Evans

September 2015: First Edition

Revision History for the First Edition
2015-09-22: First Release

978-1-491-93467-8

[LSI]

Table of Contents

Preface

My first encounter with Java came as a PhD student in Spring 1998. I had been earning some extra money by helping a disabled student who couldn't physically attend his first year Computer Science classes. I'd learned Dijkstra's algorithm and enough graph theory to stay ahead of the class, and at the end of term, he came to me and asked if I'd sit in on another class for him—some new programming language called "Java."

At first I refused, but I eventually relented, and I clearly remember many a late night sitting by the Physics department printer, waiting for a print-out of some tutorials and early javadoc so I could read up before the class.

Little did I know that this language and environment would have the impact on my life and career that it has.

Acknowledgements

Thanks to my wife Anna Evans for the illustrations, to Samir Talwar, Helen Scott, and Gil Tene for technical reviews. To Dalibor Topic for correcting the timeline of events leading up to the release of OpenJDK. To the O'Reilly team: Nan Barber, Brian Foster, Heather Scherer, and Megan Blanchette. Finally to Mike, who was responsible for getting me into this industry in the first place (if you're reading this, please contact O'Reilly, and they'll reconnect us).

CHAPTER 1

History and Retrospective

The first public release of Java was May 23, 1995, as an alpha available only on Sun Microsystem's Solaris operating system. Since that first release, Java has developed into one of the world's most widely deployed programming environments. It has gained traction and market share in essentially every programming environment, especially enterprise software, server-side Web development, and mobile phone programming.

> Java is a blue collar language. It's not PhD thesis material but a language for a job.
>
> —James Gosling

It has been a long journey with many surprises along the way. That journey started in the early days with James Gosling and his team, who were sure that developers wanted advanced productivity features (such as object orientation and garbage collection), but felt that many engineers were scared off by the complexity of the languages that implemented them.

The High-Level Design of Java

In language design terms, Java has always been based on a number of deliberate, opinionated design decisions with specific goals in mind. The Java platform's initial primary goals can be summed up as:

- To provide a container for simple execution of object-oriented application code

- To remove tedious bookkeeping from the hands of developers and make the platform responsible for accounting for memory
- To remove C/C++ platform security vulnerabilities wherever possible
- To allow cross-platform execution

Notably, these goals were pursued even at the expense of low-level developer control and performance cost in the early years.

> By almost completely eliminating the cost of porting, Java allowed developers to focus on solving business problems.
>
> —Georges Saab

The portability goal was enthusiastically referred to as "Write Once, Run Anywhere" (WORA). It represents the idea that Java class files could be moved from one execution platform to another and run unaltered. It relies on the existence and availability of a Java Virtual Machine (JVM) for the host platform. Porting Java to a new platform thus becomes a matter of implementing a JVM that will run on the new platform in accordance with the virtual machine specification (usually called the VMspec).

A Brief History of Java

The world that Java arrived into was very different to the one we live in today. Microsoft was counting down to their August release of Windows 95 (which would launch without a Web browser). Netscape was yet to go public (IPO) on the NASDAQ exchange, although its browser was steadily gaining in popularity. The Internet, as a whole, was yet to truly enter the public consciousness.

As a harbinger of things to come, Sun's initial release of the Java language was accompanied by HotJava, a browser intended to compete with the then-dominant Mosaic browser, and to supercede it via the introduction of richer, dynamic applications using Java's applet technology. Despite this, few could have predicted the impact that Java's seemingly modest release would ultimately have.

Rapidly growing public interest in the Internet, fueled by Netscape's IPO and other market events, produced conditions that kicked off a first wave of enthusiasm (and more than a little hype) for Java. This would ultimately lead to some surprising consequences, not least of which was the renaming of an unrelated scripting language to "Java-

script" in order to cash in on the public profile of the Java ecosystem.

Since those early, heady days, Java has had a fairly conservative design philosophy and an often-mocked slow rate of change. These attributes, however, have an overlooked flip side—a conscious attempt to protect the investment of businesses that have adopted Java technology.

> After the USENIX session in which James Gosling first talked publicly about Java, people were already dancing on Java's grave.
>
> —Mike Loukides

Not only that, but Gosling's vision has been more than vindicated—the design decisions of Java's early days are now considered uncontroversial. This provides a clear example of the old software maxim that "the most hated programming languages are inevitably the most widely used." In Java's case, that extends to the plagiarism of Java's ideas and design principles.

As one example, very few application developers would even try to defend the opinion that memory should be managed by hand these days. Even modern systems programming languages, such as Go and Rust, take it as a given that the runtime must manage memory on behalf of the programmer.

The Java language has undergone gradual, backwards-compatible revision but no complete rewrites. This means that some of Java's original design choices, made out of expediency due to the restrictions and conventions of late 90s technology, are still constraining the platform today.

The early years of the 21st century saw the rise of Enterprise Java, pushed heavily by Sun (and later Oracle) as the future way to develop applications. Initial versions of the platform (known originally as J2EE, and more recently as Java EE) were criticized for performance and complexity problems. However, despite these issues, the use of Java in enterpises continuedcto grow at a rapid rate.

Eventually, more lightweight frameworks (e.g., Spring) emerged. As these challengers added features, they inevitably grew in size and complexity. At the same time, the Java EE standards progressed, shedding unecessary layers of configuration, and began to focus on the core needs of developers inside the enterprise.

Today, the enterprise space remains vibrant, with healthy competition between different frameworks and approaches to large-scale development. In the last 10 years, only Microsoft's .NET framework has offered any serious competition to Java for enterprise development.

Meanwhile, the core Java platform (the "Standard Edition," or Java SE) was not standing still in the early part of the 2000s. Java 5, released in 2004, was a significant milestone, and introduced major changes to the core language. These included generic types, enumerated types, annotations, and autoboxing.

The standard library and core application programming interfaces (APIs) were also substantially upgraded—especially in the areas of concurrent programming and remote management (both for applications and for the JVM itself).

These changes were judged to be a huge step change in Java's evolution, and no release until Java 8 would have the same impact on the Java world. Sun also finally dropped the "Java 1.X" naming scheme, and started using the major number instead, so that this release was Java 5.

The waves of change, from language changes in Java 5, to low-level technical upgrades such as on-demand or Just-In-Time (JIT) compilation (Java 1.3), through to procedural and standardisation structures, such as the Java Community Process, or the Java Language Specification, have carried Java forward as a language, platform, and ecosystem.

History of Open-Source Java

This can be seen through the evolution of Sun's (and later Oracle's) attitude to community and open-source. Despite declaring huge support for open-source, Sun proceeded cautiously where Java's intellectual property was concerned.

The Java Community Process (JCP) was set up by Sun in 1998 as a way to bring other companies and interested parties into the development process for Java standards. Sun wanted to bring potential rivals "inside the tent," but didn't want to lose control of Java in the process. The result was an industry body that represented a compromise between competing companies that still had common cause.

Responding to pressure from the market and the wider community, Jonathon Schwartz (then CEO of Sun) announced the open-sourcing of Java live on stage at JavaOne 2006. Legend has it that this announcement was done without the full knowledge of his management team. This led to the creation of the OpenJDK (Open Java Development Kit) project in 2007, which is still responsible for the development of the reference implementation of the Java platform today.

Sun now had both a somewhat-open standards process, and an open-source reference implementation for Java. However, the path to an open Java was not completely smooth. The release train for Java 6 was already well underway, and it was felt to be too difficult to try to move Sun's toolchain and development practice over to an open process. Instead, a drop of code from the in-development Sun proprietary JDK 7 was taken, scrubbed, and released as the seed for OpenJDK 6.

The release was further complicated by Sun's decision to exempt certain components from the open-source release, citing problems in obtaining agreement from the copyright holders.

Due to this approach, OpenJDK 6 never had a release that precisely corresponded to a Sun JDK release. However, the release train for JDK 7 fixed this process, by quickly moving to an open process and by having all normal commits be made directly into the open repositories from then on. This process is still followed today.

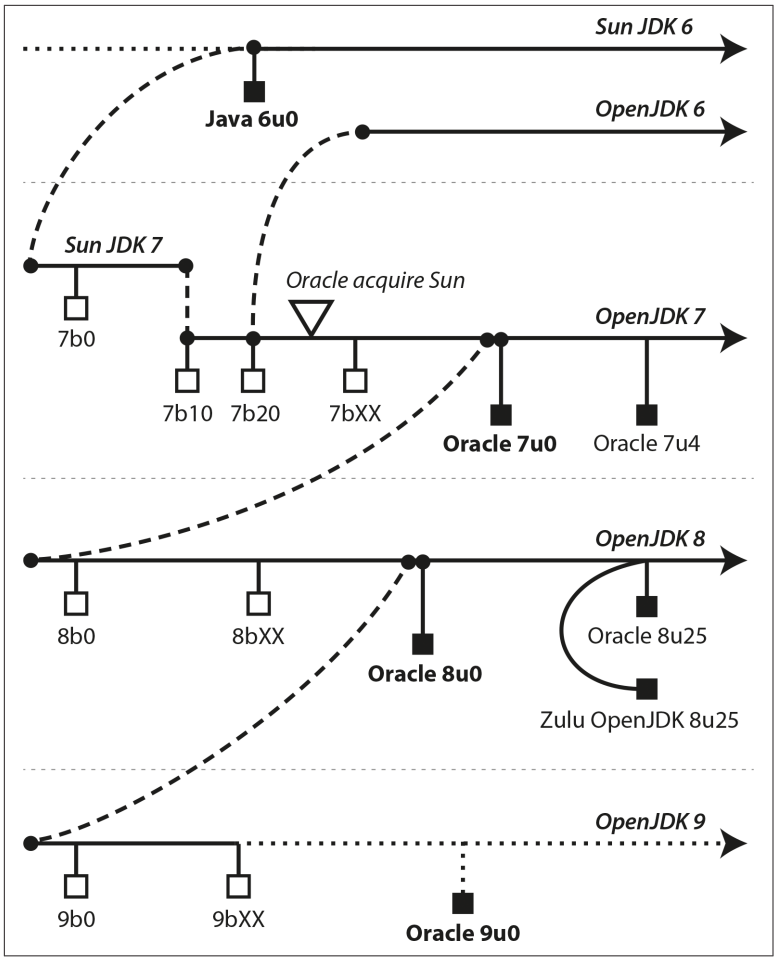

Figure 1-1. The OpenJDK branching process

For some participants in the open Java community, Sun did not go far enough in open-sourcing the platform. For example, the Testing Compatibility Kit (TCK) was not opened, and remained proprietary software. This presented a hurdle for non-Sun implementations of Java, because an implementation must pass 100% of the TCK in order to be certified as a compatible implementation of Java.

The Apache Foundation started an open-source, cleanroom implementation of Java in May 2005, and by the autumn of 2006 it had become a top-level Apache project called Apache Harmony. When the project was underway, Apache approached Sun for a TCK.

While not outright saying "No," Sun dragged its heels over the request. This was particularly true with respect to the "Field of Use" restrictions that Sun had added to the Java SE TCK license, which prevented Java SE from being legally run on a mobile phone.

Finally, on April 10, 2007, the Apache Foundation sent an open letter to Sun demanding an open-source license for the testing kit that would be compatible with the Apache license, rather than the GNU Public License (GPL) that OpenJDK uses. Sun did not produce a suitable license, and the dispute rumbled on.

Harmony would go on to be used as the basis for Google's Android application programming framework for mobile devices. As of 2015, it is still the primary standard library for Android developers.

After the release of Java 6 in December 2006, Sun embarked on an ambitious plan for Java 7, which was to grow as time passed, so that by late 2009 it included a proposal for some much looked-for major new features.

These included:

- Lambda expressions, and with them a slightly more functional style of programming
- A completely new system of modularising and packagaing Java classes to replace the aging Java Archive (JAR) format that was based upon ZIP files
- A new garbage collector called Garbage First (G1)
- Enhanced support for non-Java dynamic languages running on the JVM

Many in the community began to question when, if ever, version 7 would actually arrive. However, they were soon to discover that Sun (and Java with it) potentially had far bigger problems than just an endlessly slipping release date.

The Age of Oracle

By 2009 it was clear that Sun's revenue and profits were in serious trouble, and Sun's Board began to seek buyers for the business. After talks with IBM (and allegedly HP) failed, Oracle emerged as a buyer, and made a substantial offer in the spring of 2009. Upon obtaining

approval from the US and EU governments, the acquisition of Sun was completed on January 27, 2010.

The first few months following the acquisition contained a few upsets and high profile departures from Oracle (including James Gosling, the original inventor of Java). Outside of the Java space, Oracle fared badly, and was seriously criticized for its handling of some of the open-source technologies they had acquired from Sun.

In the Java space, trouble was to arrive late in 2010 when negotiations between Oracle and Google over licensing the Java technology used in Android broke down, and Oracle sued Google for copyright and patent infringement.

The ecosystem was to receive another shock, when Apache officially withdrew its membership of the JCP after Oracle refused to grant Apache a TCK license for Harmony under acceptable terms, despite having supported Apache against Sun in the original dispute. The Harmony project struggled on until November 2011 before finally choosing to disband, with Android being the only real inheritor of the technology.

Apache has not, as yet, returned to the JCP, but many of Apache's projects are implemented in Java or other JVM languages. The Oracle/Google lawsuit was to rumble on until May 2012, when the judge found decisively in favor of Google (although some minor aspects of the case continue in the appeals courts).

With the effective end of the Oracle/Google lawsuit and the initial uncertainty regarding Oracle's stewardship of Java fading, the Java platform seems to have settled into a period of relative calm.

On the engineering side, this has been seen in several places—most clearly in that the monolithic plan for Java 7 was cut into more manageable pieces. First, some relatively simple features and internal engineering work were shipped as Java 7 in July 2011. Next, the lambda expressions and functional programming support followed as part of Java 8 in March 2014. Finally, the long-delayed modularity support is intended to arrive as part of Java 9, which is expected to be released in September 2016.

Retrospective

To conclude this chapter, let's try to put some of this history into context, remembering of course that hindsight is 20/20. Nowhere is this more apparent than with Moore's law, the hypothesis made by Intel founder Gordon Moore to the effect that: "The number of transistors on a mass-produced chip roughly doubles every 2 years."

This phenomenon, representing an exponential increase in computer capability since 1965, has had transformative effects on computing (and knock-on effects to human society). Java has been a particularly fortunate beneficiary of the availability of an increasing amount of computer power.

> Software is eating the world.
>
> —Marc Andreessen

Long term bets in software have always been notoriously difficult. For example, the decisions made by many national governments to standardize on Microsoft in the early 90s proved to be exceptionally expensive, and provided a very powerful advantage to the vendor in winning additional work.

Java has benefited from some early design decisions, that could be seen either as prescient or very lucky. The WORA approach provided a platform-neutral approach at a time when few predicted that the Intel family of chip designs would become dominant across essentially the entire industry. It also protected Java from being locked in to a single operating system.

Moore's law also aided Java by providing a free lunch in terms of computing capability. The processor intensive aspects of the modern Java platform, such as Profile Guided Optimization (PGO) and JIT compilation, are only possible because of the incredible growth in power of processors.

The Java design principle of backwards compatibility has also proved to be very helpful to Java. Large companies have felt confident about investing in Java as a platform in part because of the visible commitment to maintaining older versions and not breaking working code by platform upgrades.

Java has also been less fortunate in some areas, partially because of its early success. Many large, slow moving organizations (including

more than a few governments) adopted Java applets as a standard technology for accessing systems over the Web.

Applets are a technology that are very much of their time, and they have not aged well. The technology proved to be very difficult to evolve, and so applets have not been considered to be a modern development platform for many years now. However, they doggedly persist due to some large early adopters being very resistant to change. The resulting effect to the ecosystem is that Java applets are still present in the platform, and are a major contributor to security problems.

As we reach Java's 20th birthday, some observers have begun to talk of "the second age of Java." Java's high performance, stability, and large numbers of developers are making it an attractive choice for many software projects. Not only that, but many of the most interesting new languages (such as Clojure and Scala) are implemented on top of the JVM.

Will Java be around in a recognisable form for its 30th (or 40th) birthday? The future is, of course, uncertain, but on the current evidence, it seems entirely possible.

CHAPTER 2

The Java Language

In this chapter, I'll discuss the Java language, its evolution, and its status after 20 years in the field. As you'll see, the language has withstood the test of time pretty well.

Primary Java Language Design Goals

Java has a number of design goals that have formed part of the platform since the earliest days.

Backwards Compatibility

The Java platform has always been very concerned with backwards compatibility. This means that code which was written (or even compiled) for an earlier version of the platform must continue to keep working with later releases of the platform.

This principle allows development groups to have a high degree of confidence that an upgrade of their JDK or JRE will not break currently working applications. The platform has been very successful in achieving this and code written for Java 1.0 will still compile 20 years later on a Java 8 installation without modification (in fact, compatibility is even stronger than that, as you'll see in the next chapter).

Backwards compatibility is a great strength of the Java language and platform, but in order to achieve it some necessary constraints are required. You'll meet a good example of this presently, and see in

detail how it impacted the evolution of the most recent version, Java SE 8.

Easy to Learn and Read

Java's approach to code is that it should be easy to read, in accordance with the well-known principle that code is read more often than it is written.

> Java feels very familiar to many different programmers because we preferred tried-and-tested things.
>
> —James Gosling

By preferring familiar things, the Java language aims to be easy to learn and to teach. It should be no surprise that the language is one of the mostly widely used teaching languages in universities.

The Java language also promotes the old adage of "pity the poor maintanence programmer." If code is read more often than it is written (and it is), then comprehension is key when reading source code. This is especially true when reading code due to an outage or a bug.

Java serves this goal by providing a lot of information, especially about the type of data or objects, in the source. Some users find this information to be overly verbose or repetitive, but it can be very helpful, especially to newcomers, or when debugging.

Modern development environments (IDEs) also alleviate the need for a lot of boilerplate, by auto-generating code where possible. Still, critics argue that it should be unnecessary for tools to work around problems like this. Whether fully justified or not, Java's wordiness has become a part of the folklore surrounding the language, to the extent that developers who come to Java from other languages frequently express surprise that the verbosity was not as bad as it had been portrayed.

Simple Type System

Java differentiates between two types of values. These are object references, and the eight primitive types (boolean, byte, short, char, int, long, float, and double), which are not objects but merely immutable data items.

This split is relatively simple, with the type system for objects being single-rooted (everything ultimately inherits from `java.lang.Object`) and single-inherited (every class has only one parent class).

The split between references and primitives in Java's type system represents something of a compromise and an accident of history.

The distinction between objects and primitives made a certain amount of sense in the late 1990s when Java was developed, as performance of basic numerical operations was an important design goal. However, as the industry developed in the early 21st century, languages like Ruby and Scala proved the value of "everything is an object." Compiler technology developed to allow specialization of objects to primitives where possible, providing the convenience of object-oriented programming with the performance of primitives.

Unfortunately, this was largely too late for Java, as backwards compatibility of the language and type system, and the sheer volume of existing Java code made moving to "everything is an object" simply impractical. Java instead implemented a partial solution—the automatic "boxing and unboxing" of primitive values to objects belonging to a specific partner class of each primitive type. This was reasonably successful, but came at the cost of additional complexity and also exposed previously rather obscure debates about object identity, and forced many more programmers to think about them.

In Java's reference type system, the approach to object-oriented programming started out reasonably simple. All classes have but a single parent class, and all classes ultimately inherit from `java.lang.Object`. As this form of inheritance is rather inflexible, Java 1.0 also introduced a separate concept, the interface, which provides a specification for functionality that a class may advertise that it offers.

If a class wants to declare that it offers functionality compatible with an interface, it does so via the `class Foo implements Functionality` construct. In Java 1.0 until Java 7, the class was then required to provide implementation code for every method declared in the interface. Interface files were not allowed to provide any method bodies—merely signatures and names of methods that formed part of the collection of functionality that must be implemented to conform to the interface.

This model of types was significantly extended in several different directions in Java 5. Firstly, the notion of the typesafe constant, or enumeration was introduced via the `enum` keyword. This enabled developers to indicate that a particular type only had a finite and known number of possible, and constant, values. This replaced the C/C++ convention of using integer constants via the `typedef` mechanism.

Java's enum constants are stronger than predecessors because they are true types that inherit from `Object`. This facility relies upon additional language machinery to ensure that only the specified instances can ever exist.

Enums were a very useful addition to the Java type system, and represented a significant upgrade to previous approaches. However, it is important to note that compared to Scala and other languages with more advanced type systems, Java only allows the representation of disjoint alternatives as instances, rather than as types.

The second addition to the type system introduced in Java 5 was a metadata facility. Java developers had previously used "out of band" information, such as naming conventions (JUnit) or additional "marker" interfaces (that contained no methods). Java 5 introduced annotations, as a way of exhibiting additional metadata which, while relevant, was essentially independent of the functionality of the type. This is sometimes referred to as "orthogonal" type information.

NOTE

Annotations proved to be a fundamentally different and new part of Java's type system.

Annotations are fundamentally more flexible than interfaces (although the mechanism makes use of them). Instead, classes, methods, and even packages can be annotated with additional information. For example, if a method is intended to be the end point for a web services (REST) call, then appropriate information and support can be provided automatically by the web container hosting the service.

Language and VM Initially Influenced Each Other

In the earliest days of the platform, the designs of the Java language and the JVM each exerted a certain degree of influence on the other. This can be seen, for example, by the typing of Java bytecodes.

The Java language has separate primitive and reference types (which point at objects contained in Java's heap). This means that a programmer's code should always know the type of any expression. This manifests itself in the JVM—specifically in the principle that the interpretation of any given bit pattern differs, depending on whether the pattern has the type of an `int`, a `float`, or a heap address.

Accordingly, JVM bytecode is typed and the types are essentially those of the Java language. The result is that most JVM bytecodes fall into "families" that have several individual opcodes that deal with the specific circumstances that are encountered in normal coding.

For example, the return family of bytecodes contains `ireturn` for a method that returns an `int`, `dreturn` for a method that returns a `double`, and `areturn` for a method that returns a reference (think: "address").

In recent years, the Java language has moved from being deeply connected with the workings of the JVM, to being "first among equals" —merely the first high-level language that ran on the JVM.

The most obvious sign of Java resigning its privileged place in the firmament of JVM languages came with Java 7, when the `invokedynamic` bytecode was added to the standard. At that time, the reference implementation Java language compiler, Oracle's `javac`, would not under any circumstances emit an `invokedynamic` opcode. Here was an opcode that had been added purely for the benefit of non-Java languages, with no reference to Java whatsoever.

Libraries

Not everything in language design is the language core. Many features can be implemented in a variety of ways, and language change is expensive in terms of resources. Therefore, wherever possible,

Java has a principle that new features should be implemented as libraries.

> Java itself, as a language, is pretty simple, as are most languages. The real action is in the libraries, and we tried hard to have a fairly large class library straight out of the box.
>
> —James Gosling

One place in which this principle manifested itself was in the area of arrays and collections. All programming languages need to manipulate data en masse, and all languages provide an assortment of fundamental data structures for the programmer to use as part of the basic installation. Java is no exception.

Arrays and Collections

Java 1.0 had two different types of data structures that shipped with the platform—arrays, which were provided within the syntax of the language, and some classes (such as `Hashtable` and `Vector`) that resided within the `java.util` package of the Java standard library.

This represented a compromise. Programmers expected a familiar C-like syntax for handling arrays, especially arrays of primitive types. On the other hand, as heap-managed data, arrays were definitely objects in the Java worldview. This resulted in a halfway house, where the object nature of arrays was somewhat sidelined, and set up a rather obvious disconnect between seemingly simple arrays and the object-oriented view contained in the classes of `java.util`.

With the arrival of Java 1.2 (or Java 2 as the marketing went), Java received a major upgrade in the form of the Java Collections libraries. This was a full set of data structures covering the most common cases, such as lists, sets, and maps. However, with the increased emphasis on the object-oriented approach, the split between arrays and object-based data structures became even more obvious.

One of the biggest changes to the language was Java Generics, which appeared as a part of Java 5. Generics enable the programmer to represent a composite type, usually thought of as comprising a "container" and "payload" type. Before Java 5, the programmer could only discuss a type as `Box` without any reference to what the box contained. Using Java Generics, however, the type could be further qualified to `Box<Egg>` or `Box<Wine>` or `Box<Shoe>`. This provides for

improved safety when programming, as the source compiler can now detect mistaken attempts to put wine into objects that are really shoeboxes or eggboxes.

Recent Developments

The last few releases of Java have seen some steps in the direction of making Java require less boilerplate code.

Reducing Verbosity—Java 7

First, in Java 7, the notion of type inference was introduced. Previously, `javac` would require the poor programmer to write out type information in excruciating detail:

```
Map<String, String> enToFrDict = new HashMap<String, String>();
```

The arrival of Java 7 allowed programmers to make some modest savings of keystrokes:

```
Map<String, String> enToFrDict = new HashMap<>();
```

with the type information of the generics being ommitted on the right-hand side.

Another place where the type information was used to condense code was in the feature called *multicatch*. Java contains a concept of an exception, which is used to indicate that an unexpected condition has been encountered. Operations such as file handling may result in a number of different possible error conditions (file not found, permission denied, etc.) occuring. Java therefore allows multiple recovery strategies (called "catch blocks") to be specified, with the strategy executed chosen based on the type of problem experienced.

In Java 6 and earlier, each catch block must be specified separately, possibly leading to a lot of repeated code:

```
try {

   // Try to read a class file from disc, classload it to
   // get a class object and then access a method reflexively.
   //
   // This process can fail in a number of different ways...

} catch (IOException iox) {

    // ...
```

```
} catch (InstantiationException instx) {

    // ...

} catch (IllegalAccessException ilax) {

    // ...

} catch (NoSuchMethodException nsmx) {

    // ...

} catch (SecurityException secx) {

    // ...

} catch (IllegalArgumentException ilargx) {

    // ...

} catch (InvocationTargetException invtx) {

    // ...

}
```

In Java 7 however, the language syntax was extended to allow a catch block to handle several different exceptions, like this:

```
try {

    // Same loading process

} catch (IOException | InstantiationException
| IllegalAccessException | NoSuchMethodException
| SecurityException | IllegalArgumentException
| InvocationTargetException ex) {

    // But now we can handle all failures with a
    // single recovery block

}
```

Towards the Future—Java 8

One of the most eagerly awaited features of Java 8 was the addition of language support for lambda expressions (aka function literals or closures). Previously, Java developers had been forced to use anonymous classes as a verbose substitute. However, simply adding the language syntax was not the whole of the story.

The overall aim was not for lambda expressions per se, but rather to evolve Java's collections to allow support for "more functional" operations, such as map and filter. This was problematic, due to Java's requirement for backwards compatibility.

One of Java's language constraints that arises from compatibility is that Java interfaces may not have new methods added to them in a new release of the interface. This is because if methods were to be added, existing implementations of the interface would not have those methods. That would mean older implementations would not be seen as a valid implementation of the new version of the interface, and so binary backwards compatibility would be broken.

This could not be allowed to happen, and so a different path was chosen—allowing an interface to specify a default implementation for new methods. With this change, new methods can be added to interfaces—provided that they are default methods. Older implementations of interfaces, which do not have an implementation of the new method, simply use the default provided by the interface definition.

This change had the side effect of changing Java's model of object-oriented programming. Before Java 8, the model was strictly single-inherited (for implementation) with interfaces providing an additional way for types to express their compatibility with a capability. This was sometimes criticized for forcing types to repeat implementation code unecessarily.

With Java 8, however, Java's model of objects changes, to allow multiple inheritance, but only of implementation. This is not the full multiple inheritance of state as implemented by C++ or Scala. Instead, it can be thought of as essentially being a form of "stateless trait" or a way to implement the mixin pattern.

Java as a language has evolved fairly gradually, but the experiences of adding lambdas (sometimes called Project Lambda after the working group that produced it) have shown us that it is entirely possible for major changes to be implemented without giving up backwards compatibility or the "feel" of Java.

Java's Greatest Hits

Like most languages, Java has good parts and bad parts. Some of Java's successes are particularly notable and have been responsible

for making Java one of the pre-eminent programming languages of the world.

Threading

Java was the first mainstream language to support threading from day one. This opened the door to concurrent programming becoming a part of the mainstream of developers working lives.

The launch of Java coincided well with the growth of multicore systems. Threaded programs can scale out to multiple cores in a way impossible for single-threaded code.

Threading proved to be an essential tool when developing larger and more sophisticated systems, from desktop to server environments. Java's ground-up support for it contrasts favorably with languages in which threading was an afterthought. For example, Perl's support for threading never really became stable, and in the Ruby world, most groups who want to use threading prefer the version of Ruby that runs on top of the JVM and uses the Java threading support (JRuby).

Language Stability

The core Java language has evolved only modestly since the first version. There are only a few keywords that have been added since the earliest versions, and the language has remained quite recognizable over time, with only the additions of generics (Java 5) and lambdas (Java 8) really changing the feel of the language.

This has been one of the features that has made Java so popular with enterprises and larger shops. The stability has made it possible to build engineering teams with a clear roadmap of the technology, and further speaks to the aspiration of Java to be a technology in which ordinary developers can produce business functionality.

Java's Type System

Java's type system can be characterized as:

- Nominal—The name of a Java type is of paramount importance. Java does not permit structural types in the way some other languages do.
- Static—All Java variables have types which are known at compile time.

- Object/Imperative—Java code is object-oriented, and all code must live inside methods, which must live inside classes. However, Java's primitive types prevent adoption of the "everything is an object" worldview.
- Slightly functional—Java provides support for some of the more common functional idioms, but more as a convenience to programmers than anything else.
- Modestly type-inferred—Java is optimized for readability (even by novice progammers) and prefers to be explicit, even at the cost of repetition of information.
- Strongly backwards compatible—Java is primarily a business-focused language, and backwards compatibility and protection of existing codebases is a very high priority.
- Type erased—Java permits parameterized types, but this information is not available at runtime.

Java's type system has evolved (albeit slowly and cautiously) over the years and, with the addition of lambda expressions, is now on a par with the type systems of other mainstream programming languages. Lambdas, along with default methods, represent the greatest transformation since the advent of Java 5, and the introduction of generics, annotations, and related innovations.

The Java Hall of Heroic Failure

Java has been has been a major part of the computing landscape for most of the last 20 years. In that time, along with major successes, there have been a number of experiments in language design that have not gone as well as had been hoped. Long-lived languages inevitably have warts and annoyances upon them. For example, in Java's case:

Java Beans

This is the idea that every field on every Java object should obey a public getter and setter convention, in order to promote interoperability and a nebulous and poorly-conveyed idea of standardization. They were originally felt to be the right convention for handling most Java objects.

NOTE

The term "Java Beans" predates Enterprise Java, and they should not be confused with Enterprise Java Beans (EJBs).

In practice, however, the idea that all properties should be mutable turned out to be toxic. Instead, it became clear that concurrect programming contained a can of worms that made the Java Beans approach unsound for most usages.

Finalization

Probably the worst feature in Java. The original intention was to provide a method for automatically closing resources when they were no longer needed (in a similar spirit to the C++ RAII pattern). However, the mechanism relies upon Java's garbage collection, which is non-deterministic. Thus, using finalization to reclaim resources is fundamentally unsafe (as the developer does not know how long it will be until a resource is freed). Therefore it is impossible to use finalization as a way of avoiding resource exhaustion, and the feature cannot be fixed. In other words, never use finalization.

Instead, Java 7 introduced "try-with-resources" as a way of automatically controlling and closing resources that actually satisfies the needs of Java programmers.

Java EE Over-Configuration

Early versions of the Java EE standards required the programmer to write an almost overwhelming amount of configuration, mostly in XML. This almost completely obscured the goal of the platform, which was to provide a simple, business-focused environment in which infrastructure concerns could be provided by the container.

Over time, successive versions of Java EE (as it is now known) reduced the configuration burden, to the extent that Java EE7 is a completely modern and pleasant environment for serverside web development. However, the mention of earlier versions of Java's Enterprise Java Beans still provokes horror from many older developers.

Threading

Java was the first mainstream language to support threading from day one. As a result, it was the proving ground where many of the practical problems involved in writing safe, robust concurrent code were discovered.

`Thread` is a very low-level abstraction, and programmers were expected to manually manage concurrency in the earliest versions of Java. This improved with the arrival of `java.util.concurrent` in Java 5, and its subsequent development. However, by that time, many programmers had felt firsthand the complexities and frustrations of programming with Java's threads.

Other languages have learned from the pain felt by the early pioneers of concurrent programming in Java. For example, Scala and Clojure have both built more sophisticated and safer constructs on top of the underpinnings provided by Java and the JVM. Elsewhere, the actor model and Go's goroutines have provided an alternative view of concurrent programming.

Conclusion

Software engineering is a profession that is still, despite 50 years of practice, a very immature discipline. Each new language that wants to be successful should strive to push the envelope of what is known about software and the process of its creation. Some pushes will succeed and lead to valuable new insights into how programmers should think about the complex domains that exist within software.

Don't underestimate the value of failed experiments, though. Any adventurous language with a significant userbase over an extended period should have plenty of warts and battle scars to its name, and Java is no exception.

CHAPTER 3

The Java Virtual Machine and Platform

The Java language drew upon many years of experience with earlier programming environments, notably C and C++. This was quite deliberate, as James Gosling wanted a familiar environment for programmers to work within. It isn't too much of an exaggeration to describe the Java language as "C++ simplified for ordinary developers."

However, Java code cannot execute without a Java Virtual Machine (JVM). This scheme provides a suitable runtime environment in which Java programs can execute. Put another way, Java programs are unable to run unless there is a JVM available on the appropriate hardware and OS we want to execute on.

This may seem like a chicken-and-egg problem at first sight, but the JVM has been ported to run on a wide variety of environments. Anything from a TV set-top box to a huge mainframe probably has a JVM available for it.

In environments like Unix and Windows, Java programs are typically started by from the command line, e.g.:

```
java <arguments> <program name>
```

This command starts up the JVM as an operating system process. In turn, this process provides the Java runtime environment, and then finally executes our Java program inside the freshly started (and empty) virtual machine.

The Design of the JVM

The design of the JVM also drew on the experiences of its designers with languages such as C and C++ (but also more dynamic languages such as Lisp and Smalltalk). In addition, it took some bold steps to advance the state of the computing industry. These steps included the use of stack-based virtual machine technology to assist porting and to enable a strong security "pinch point."

When the JVM executes a program, it is not supplied as language source code. Instead, the source must have been converted (or *compiled*) into a form known as Java bytecode. The JVM expects all programs to be supplied in a format called *class files* (which always have a `.class` extension). It is these class files, rather than the original source that are executed when a Java program runs.

The JVM Interpreter and Bytecode

The JVM specification describes how an interpreter for the bytecode must operate. Put simply, it steps through a program one bytecode instruction at a time. However, as Java and other JVM languages natively support threading, both the JVM and the user program are capable of spawning additional threads of execution. As a result, a user program may have many different functions running at once.

The Java language and JVM bytecode have developed somewhat separately, and there is no requirement for the two to exactly replicate each other. One obvious example of this is what happens to Java's loop keywords (`for`, `while`, etc.). They are compiled away by `javac`, and are replaced with bytecode branch instructions. In fact, in JVM bytecode, the flow control instructions consist of if statements, jumps, and method invocation.

From the bytecode perspective this is also a safety feature, as it partitions transfer of control into local operations (essentially just `if` and `jmp`), which can be range-checked, and non-local operations, which are forced to go through the method dispatch mechanism. Nowhere in JVM bytecode is C's unsafe "transfer control to arbitrary memory address" construct supported.

Bytecode also allows a number of perfectly legal constructions that no Java source compiler would ever emit. However, if we write byte-

code directly we can access these capabilities and create classes with some unusual properties.

For example, the Java language spec requires that every class has at least one constructor, and `javac` will insert a simple constructor if it has been omitted. However, in raw bytecode it is completely possible to write classes that have no constructor. Such classes will be completely usable from Java, provided only static access to fields and methods is used.

The separation was not required by either language or the JVM, but the co-evolution of both aspects means that there are areas where the nature of the JVM "leaks through" into the Java language, and vice versa.

Influence of Language and VM on Each Other

For example, consider the insistence of the Java language that `void` is not a type, but merely represents the absence of a return type. This outlook can seem strange to the Java beginner, but it really stems from the design of the JVM.

The Java Virtual Machine is a stack machine, in the sense that each method has an evaluation stack in which intermediate results are worked out before a final result is handed back to caller. To see this in action, consider this bit of Java code:

```
class GetSet {
    private int one;

    public int getOne() {
        return one;
    }

    public void setOne(int one) {
        this.one = one;
    }
}
```

This Java code, when compiled with `javac`, produces this bytecode for the `getOne()` method:

```
  public int getOne();
    Code:
       0: aload_0
       1: getfield      #2                  // Field one:I
       4: ireturn
```

When executed, the `aload_0` bytecode places `this` on the top of the execution stack. Next, the `getfield` opcode consumes the value of the top of the stack and replaces it with the object field that corresponds to position 2 in the table of constants (the class's Constant Pool) of this class.

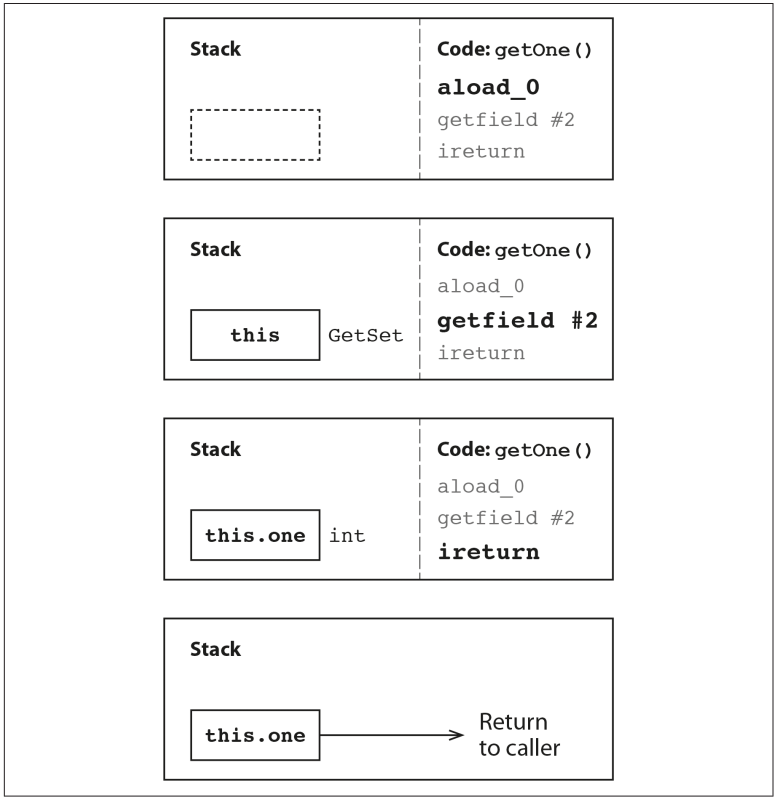

Figure 3-1. The JVM stack for getOne()

Finally, the method explicitly returns to caller, indicating (by the initial letter of `ireturn`) that there is an `int` at the top of the stack that should be collected as the return value. This explicitness of return type allows for more static checking of JVM bytecode during classloading, and helps to improve the Java security model.

Now consider the corresponding setter method. This compiles to bytecode as shown:

```
public void setOne(int);
  Code:
```

```
0: aload_0
1: iload_1
2: putfield      #2                  // Field one:I
5: return
```

Here, of course, there is nothing to return, as the `putfield` opcode consumes not only `this` but also the value that had been pushed onto the stack above it (the value that the object field was to be set to). Accordingly, the return opcode has no prefix—as the evaluation stack of `setOne()` is entirely empty.

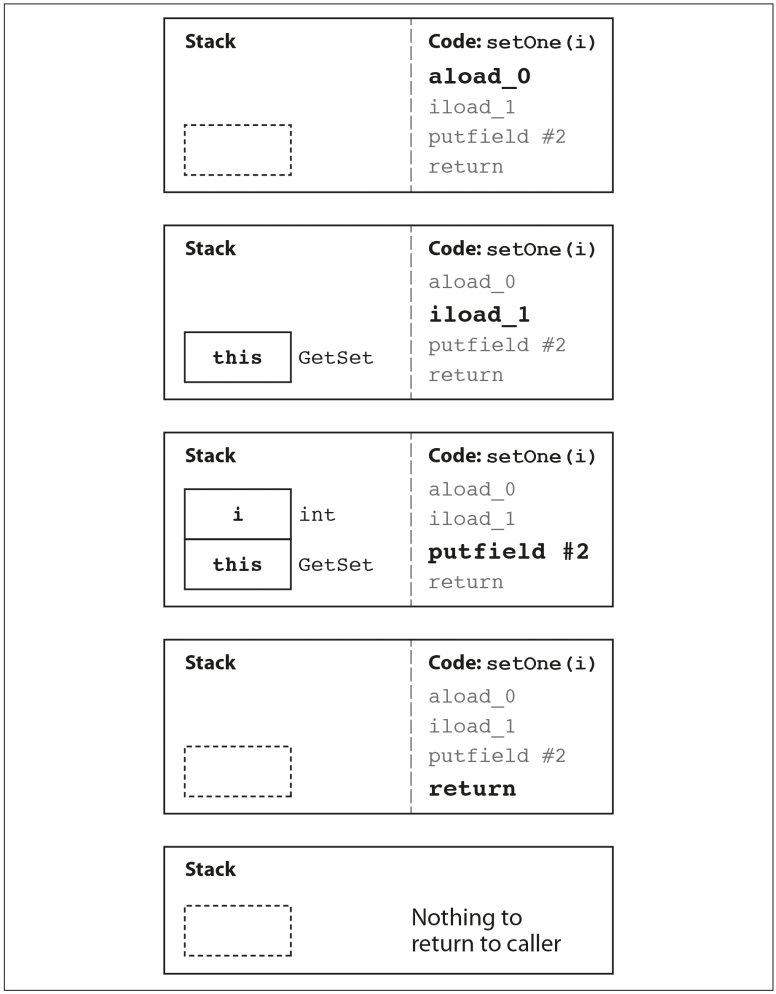

Figure 3-2. The JVM stack for setOne()

So, the Java keyword `void` indicates that the "method returns no value," and it's not a type, because it corresponds to the JVM condition of "method returns with the evaluation stack in an empty state."

The low-level JVM design decision here mirrors the condition of the method execution stack in the signature of the high-level language. Java makes this decision in preference to alternatives, such as creating a specific type for the purpose of indicating this condition (as Scala does with its `Unit` type). This design decision has some far-reaching consequences, especially when more advanced Java language constructs (such as reflection and generics) are introduced; that's why language design is a complex undertaking. To define a language feature in any given release is to open the door to unintended consequences in the future, when additional language features are seen as desirable.

Due to Java's stringent backwards compatibility requirements, these unknown interactions between language features (present and future) are a force that has driven the language to be very conservative. If the language has to maintain perfect consistency when adopting new features, then today's must-have new feature may be the exact same thing that constrains the language on the next release.

For example, Java 7 introduced the `invokedynamic` bytecode. This was a big step in advancing the JVM and making it friendlier to non-Java JVM languages. It was introduced into the platform very cautiously. The version of `javac` that ships with Java 7 will not, under any circumstances, emit an `invokedynamic` instruction. For Java 8, the feature is used to implement features related to lambda expressions (such as default methods), but there is still no direct language support for manipulating dynamic call sites.

Self-Management

There is another major aspect of the JVM's design that's not always recognized by beginners: the use of runtime information to enable the JVM to self-manage, sometimes called *profile guided optimization* (PGO).

Software research had revealed that the runtime behavior of programs has a large number of potentially useful patterns that can't be

predicted ahead of time. The JVM was the first mainstream platform to try to utilize this research.

> Time and again when developers chose Java, they reaped the benefits of the implementation continuing to improve with the hardware and OS, making their existing programs better without their having to lift a finger.
>
> —Georges Saab

The JVM collects runtime information to make better decisions about how to execute code. Through this monitoring, the JVM can optimize a program and achieve better performance. In fact, modern JVMs can frequently provide performance beyond the capability of platforms that don't have PGO.

Just-In-Time Compilation

One example of PGO is based on the observation that some parts of a Java program will be called far more often than others (and some methods will be very rarely, if ever called). The Java platform takes advantage of this fact with a technology called just-in-time (JIT) compilation.

In the HotSpot JVM, a profiling subsystem identifies which methods of the program are called most frequently. These methods are eligible for compilation into machine code, which allows the important parts of the code to achieve far higher performance than was possible from interpreted code. In Java's 20 year history, the optimizations used by the JVM have advanced to the point where they often surpasses the performance of compiled C and C++ code.

In order to assist the JIT compiler, the `javac` source code compiler performs only very limited optimizations, and instead produces "dumb bytecode." This provides an easy-to-understand representation of the program.

For example, `javac` does not recognize and eliminate tail recursion. So this code:

```
public static void main(String[] args) {
    int i = inc(0, 1_000_000_000);
    System.out.println(i);
}

private static int inc(int i, int iter) {
    if (iter > 0)
        return inc(i+1, iter-1);
```

```
            else
                return i;
        }
```

will cause a stack overflow if run. The equivalent Scala code, however, would run fine, because `scalac` does a great deal of optimization at compile time, and will optimize away the tail recursion.

The general philosophy is that the JIT compiler is the part of the system best able to cope with optimizing code. So `javac` allows the JIT compiler free reign to apply complex optimizations (although this does not include tail recursion elimination).

Garbage Collection

The JVM allows for automatic management of memory, via *garbage collection* (GC). This typically runs as a separate, out-of-band task within the JVM, which user code neither knows nor cares about.

Java's approach to GC, at least in the Hotspot JVM, is unique. Hotspot regards collectors as pluggable systems, and out of the box the JVM has several different algorithms available. Each of these is highly configurable, and each has the ability to adapt operation to the allocation behavior and other runtime conditions of the running Java process.

The self-management features of the JVM have contributed to the emergence of highly performant execution as a defining feature of the overall Java environment. Gone are the days when Java was the punchline of jokes about poor performance. However, the JVM proved to have a reach and a utility outside of just Java code.

Beyond Java

The JVM turns out to be quite a good general purpose virtual machine. The mixture of performant primitive operations and object orientation is a good fit for a wide range of languages.

There are versions of languages such as Ruby, Python, Lisp, and Javascript that run on top of the JVM. It's relatively easy to implement a two-level interpreter, with the language interpreter written in Java. Not only that, but far more sophisticated options are possible. For example, JRuby starts off using an interpreted mode for Ruby, but then will use JIT compilation to convert important methods to

JVM bytecode. Eventually, the JVM's JIT compiler will kick in, and the Ruby method will be compiled to native code.

One of the main advantages of using the JVM as a language runtime is that it's easy to interoperate with Java bytecode. This means that each individual language need not reimplement full library support, but can start off with a language-specific wrapper over an existing library. This allows new JVM languages to piggy-back from the established Java ecosystem.

Languages that fundamentally aim to be "a better Java," such as Scala and Kotlin, require good interoperability with Java if they are to gain traction and credibility. However, what is perhaps more surprising is how good the interoperability story is for languages that are not very close to Java in linguistic terms.

For example, Java 8 shipped the Nashorn implementation of Javascript. This was the first implementation of Javascript to hit 100% conformance on the ECMA standard testing kit. Despite the historical accident that led to the similarity in names, Java and Javascript are radically different languages. The fact that Javascript can be implemented on top of the JVM is a huge win. This is further helped by the tight integration that is available between Javascript and Java.

In Nashorn, every Java type is exposed via an extremely simple and natural mechanism. This means that there is seamless access to all of Java from the scripting environment. Not only that, but every Javascript function value can be used as a Java lambda expression, and vice versa.

A very similar picture also emerges in the Clojure language. Clojure is a JVM language in the Lisp family. Language pedants may argue about whether Clojure is an actual Lisp, a Lisp dialect, or merely a language in the same overall diaspora. However, the Lisp nature of Clojure is apparent at first sight. It's not necessarily a language that would seem easy to integrate with Java at first sight, since the type systems and the feel of the languages are totally different.

In both the Nashorn and the Clojure case, the language implementors have taken time and expended effort to ensure that Java libraries are easily accessible, and that they feel idiomatic in the language that they are being transplanted into.

Since the JVM is a first-class target even for languages as different from Java as Lisp and Javascript are, it stands to reason that the JVM

would be a good general home for programming languages. There is also a historical trend—with the release of Java 7, the specifications explicitly broke the references to the Java language in the JVM spec. Instead, Java is now "first among equals" in terms of languages running on the JVM—a privileged position, but not the only game in town.

We can see this in the way that `invokedynamic` was handled. It was introduced in Java 7 in order to help non-Java languages (notably JRuby, although it would have been almost impossible to build Nashorn without something like `invokedynamic`). At the time of writing, there are no plans to give the Java language a way to directly access `invokedynamic` call sites. Instead, it's seen more as a feature for library builders and non-Java languages.

Conclusion

The JVM has been an enormous success. Some of the design lessons learned during its evolution have been widely adopted by other languages. Features that were once novel are now just part of the furniture and the standard toolkit for building virtual machines. It certainly isn't perfect, but it represents centuries of engineering effort, and the end result is a general purpose virtual machine that is arguably the best available target for all sorts of programming languages.

CHAPTER 4

Java Developers and the Ecosystem

Don't listen to Hollywood. That's good advice under most circumstances, but when thinking about technology it's particularly true. The movies would have you believing that tech is all about individual intellectual endeavor.

In fact, the practice of technology has always been a social activity. Even before mass adoption of free and open-source software, the sharing of code and ideas was already a central feature of technology culture.

So it comes as no surprise that any sufficiently advanced technology tends to develop an ecosystem and community alongside it. Once again, Java is both unexceptional, in that a developer ecosystem grew up around it, and exceptional, because that ecosystem evolved into something quite unlike any other language and platform community.

Overview of the Java Ecosystem

Java's reach encompasses, by most estimates, at least 10 million programmers. This gives it one of the largest and most important developer communities of all. Only the Web/Javascript communities (and arguably C/C++) come near to the size of the Java community.

For serious, safe application development, Java is pretty close to being the only game in town. It's a robust and stable platform, but

despite its size and reputation for solidity, Java does, at times, suffer from an image problem. Being a competent, practical language does not necessarily endear the language to people who are used to working with it all day, every day.

Another issue is that Sun's early marketing material tended to be overly optimistic in promising benefits for the developer in some areas. The Java developers who've been around since the early days are sometimes a little cynical about the platform, and may have battle scars from working with early versions of the platform.

It's also worth remembering that Java is unashamedly pitched at business and enterprise development. This isn't necessarily the same cohort of freestyling open-source enthusiasts that are sometimes found in other languages. At the risk of some stereotyping, Java developers are more likely to be the people who code during the day and go home to their families, rather than those who rush off to a hackday in the evening.

This can mean that commentators overlook the open-source parts of the Java ecosystem. On the other hand, even if Java developers are less likely to get involved in after-hours development, the sheer size of the Java community means that there are still a large number of developers engaged in open-source work in Java.

The Java Community Process

Developer uptake was good in the early years after Java's release, but Sun had bigger ambitions. They wanted Java to become a widespread standard set of technologies, but didn't have the resources to develop and support all of the integrations and components that would be required by such a broad push. As a result, Sun needed adoption of Java by larger corporations (such as IBM, Fujitsu, and HP).

Many of these companies were concerned by the rise of Microsoft, and Java technology potentially offered an opportunity to hedge that risk. Sun didn't want to cede control of Java, as it saw huge potential in the technology, and so created an industry body that resembled a standards body. The idea was that standardization would prevent the migration (followed by lock-in) of Sun's customers to Java technology stacks produced by other vendors.

Founded in 1998, the JCP is a way of formalising and standardising Java technologies. The JCP uses Java Specification Requests (JSRs), which are official working groups, led by a Specification Lead, that produce a specification document, testing kit, and a reference implementation.

The JCP is fairly unique in that it includes a patent and intellectual property regime that protects end users and participants. To participate in the JCP, corporations are required to provide a license of their patents if they are to form part of the standards.

A JSR has a defined lifecycle, whereby the technology standard is worked upon and guided through stages of maturity until it has reached the point where it is ready for widespread developer use. This should ensure that only technologies that are widely adopted enough, and have achieved a degree of acceptance and stability are targeted for standardization.

This has resulted in a process where several different classes of JSR exist. For example, each new version of Java SE, EE, and ME has an "umbrella" JSR that covers the content of the platform release. The most recent release of Java SE was version 8, and the corresponding umbrella JSR was JSR 337. However, these umbrellas usually just bring together JSRs under which major new language features have been developed. JSR 337 therefore included JSR 308 (type annotations), JSR 310 (new date and time libraries), and JSR 335 (lambda expressions).

In addition to the umbrella JSRs, and the JSRs dealing with major new features, there are also JSRs corresponding to major libraries, such as XML parsing (JSR 5) and servlets (various, latest JSR 369). There are also more niche JSRs, that cater to a particular style of programming, such as the real-time specification for Java (JSR 1). Finally, the processes of the JCP itself are specified as JSRs, so occasionally JSRs are filed to modify or update the JSR workflow or practices of the JCP.

The Independent Java Ecosystem

Java has always had an ecosystem of enthusiasts outside of Sun (and later Oracle). In the early years, developers wanted to tinker and explore the limitations of the platform, as is so often the case with open-source hackers. By bumping up the edge of the possible, devel-

opers exposed missing features that could be added into future releases, making Java even stronger.

In time, a number of independent projects evolved, and even after the open-sourcing of the platform, many developers chose to continue working outside of the official projects, such as OpenJDK.

Eclipse

IBM had been working to produce an IDE for Java, based on their VisualAge product. This led to the creation of a Java-based IDE for Java, which became known as Eclipse. In late 2001, IBM released this as open-source code, and brought together a consortium of companies to steward the technology. This led to the creation of an independent foundation, the Eclipse Foundation, in 2004.

Although the IDE product remains the principal project for which Eclipse is known, the Foundation actually hosts over 200 other software projects, covering such areas as rich client development and business intelligence and reporting.

In recent years the Eclipse Foundation has continued to grow and diversify, including to technologies unrelated to Java. It also now hosts a major project related to the emerging software technology known as the Internet of Things (IoT).

Apache

The Apache Foundation predates Java. In fact, its initial focus was the Apache web server, `httpd`. Over the years, Apache expanded outwards from the runaway success of the web server, and became a broad, language-agnostic foundation. The projects hosted under the banner of the Apache Foundation cover almost every aspect of technology where open-source code could play a role.

Not only that, but the open-source license written by the Apache Foundation was enthusiastically adopted by a large number of projects that were not part of, or governed by the Apache Foundation. It's therefore important to distinguish between "an Apache project," one that has been officially onboarded as part of the Apache Foundation, and simply "an Apache-licensed project," which just uses the Apache license.

As Java expanded into many areas of enterprise development and beyond, it was inevitable that some Apache projects would end up

being written in Java. However, at least at first, this led to the strange situation of numerous open-source libraries being written for a non-open-source platform. Even after OpenJDK became a reality, the Apache license and the GPL license used by OpenJDK remained irreconcilably incompatible.

The response of the Apache Foundation to these licensing concerns was to begin a complete compatible rewrite of the Java class libraries —Harmony. Despite being a qualified technical success, Harmony was plagued with legal problems, as discussed in Chapter 1.

This culminated in Oracle's refusal to grant a TCK licence for Harmony. Apache resigned from the JCP Executive Committee in protest in December 2010, and mothballed Harmony a few months later.

Today, the relationship between Apache and Oracle is at a near standstill. Java library projects still thrive and proliferate, both as Apache-licensed and Apache Foundation projects. However, there has been no direct rapprochement between the two sides, and the scars from the Harmony dispute are still painfully visible.

Spring

Sun had invested heavily in a bet on the rise of Java as an enterprise language and platform. The scope of the vision was quite ambitious, aiming to largely or completely free the ordinary developer of business applications of low-level concerns.

The first few versions of the Enterprise Java vision were plagued by problems. The most fundamental of these was that the problem space was simply not understood well enough at first. Separating business logic concerns from infrastructure, and separating both from configuration and deployment is a worthy goal. However, the correct positioning of the dividing lines between these concerns is somewhat more subtle than it seems at first glance.

As a result, while Java's footprint in the enterprise continued to grow, teams were looking for ways to simplify complexity and still provide more powerful techniques to define, configure, and deploy their applications.

Against this backdrop, alternatives to the official enterprise Java stacks began to emerge. One of the best known and most powerful,

was the Spring framework. This was originally created by Rod Johnson and first released as open-source software in 2003.

The initial idea behind the Spring framework was to provide a much more lightweight way of configuring and executing applications than was possible within the orthodoxy of "pure" enterprise Java. By separating out the core concern of configuration, Spring frees the container from this responsibility. The design of Spring allows the application developer to chose a container that fits the needs of the application (including not requiring a container at all).

Like any successful technology, as users became familiar with it, they started to discover use cases that were not catered for, and missing features. Over time, Spring became a larger collection of semi-related interoperable technologies that provide a full-featured framework of comparable capability to Java EE. Of course, catering to a larger set of features and concerns has its price. Spring is now no longer any smaller or less complex than the technology stack that it was originally started in response to.

The Spring community has flourished as the technology has matured, and there are now numerous Spring developers that usually, or exclusively, take the Spring stack as their baseline for any new Java development. Outside of this, Spring has permeated throughout the Java ecosystem, and most working Java programmers will have encountered some Spring technologies at some point during their career.

The Java Community Now

Today's Java community is the result of widespread developer adoption, corporate politics, global economic forces, and more than a measure of blind luck. Software development is increasingly globalized, but the simplicity and relatively small cognitive footprint of Java have meant that the platform has travelled well and prospered as the industry has expanded.

In this section, we'll look at some of the more prominent features of the global Java community.

JUGs

Java User Groups (JUGs) are informal groups of Java programmers who have chosen to organize into a loose association in order to

share experience and knowledge, network, and enhance each other's professional development.

Oracle does not enforce any particular rules on JUGs. Instead, Oracle simply asks that when a new JUG forms, they register with Oracle's community staff. The company maintains a list of groups, and offers support and promotion to them.

JUGs are a great way to meet new people, hear about new technology, expand your skills, get involved in open-source, and even find new career opportunities. Some of the largest and most influential groups include SouJava (Brazil) and the London Java Community (UK), but there are JUGs of all sizes all over the world.

One of the original maxims of free and open-source software is that all it takes is for a single developer to sit down and decide to scratch their own technical itch, and decide to share their work freely.

Developers who come from a more corporate background may not have been exposed to this philosophy as much. So they may be surprised by the small amount of work that's required to set up a JUG, collect a few Java developers together, and start making a difference.

One of the ways in which JUGs have started trying to improve the ecosystem is through the Adopt programs. These are JUG-led global programs founded by the London Java Community, and are designed to provide ways for ordinary Java developers to contribute to the development of new Java standards (JSRs) and to the reference implementation (OpenJDK). Even a single, isolated developer is welcome to participate, and can make a useful contribution (*https://java.net/projects/adoptopenjdk*).

Java Champions

The Java Champions program was started by Sun to recognize and encourage Java professionals working outside of Sun. While there's no precise definition, the core values are that a Champion should be a leader, technology luminary (both in technical stature and involvement with exciting tech), and be influential, independent, and credible to other developers.

> The Java Champions are an exclusive group of passionate Java technology and community leaders who are community-nominated and selected under a project sponsored by Oracle.
>
> —Oracle

The program contains only a few hundred expert developers worldwide, and they are a diverse group, both geographically and in every other regard. They form an informal leadership group (along with the JUG leaders) for Java as it is practised in industry.

The landscape of Java developers is complex, but remains healthy. The vast majority of Java programmers leave their work behind when they finish for the day, of course. Fortunately, the overall pool of developers is so big, that the enthusiast, or person who wants to enhance their career should find plenty of ways to engage.

CHAPTER 5

The Future of Java

Finally, let's turn to the future of the language, platform, and developer ecosystem. Increasingly, these have become interwoven, so it makes sense to treat them as a whole as we look into our crystal ball.

Java 9

The next major release of the platform is Java 9, scheduled for September 2016. As releases go, it's expected to be a fairly major one, as it contains a number of large features (although how their impact will compare to the arrival of lambdas in Java 8 remains to be seen).

Modules

If lambda expressions were the "headline" feature for Java 8, in Java 9 it is anticipated to be modules. Up until now, the largest grouping construct for Java code was a package, but the release of Java 9 will see a new concept—the module. Modules are collections of code that are larger than packages, and are no longer delivered as JAR files (which are really just *.zip* files). Instead, modules have a new file format that has been designed to be more efficient.

Modules also add a major new feature to the language, which is the ability to enforce access control across modules. That is, modules are able to fully specify their public API, and prevent access to packages that are only for internal use.

The ability for modules to allow internals access only to trusted client code will have major repercussions for Java applications. This is

most apparent in the removal of access to a class called `sun.misc.Unsafe`. This class is an internal class (as can be seen by the fact that it lives in a `sun` package, rather than a `java` or `javax` package) and should not be used directly by applications or libraries.

`Unsafe` contains functionality that enables low-level access to platform features that are normally inaccessible to ordinary Java code. It also contains code to directly access processor features, compare-and-swap hardware for example. These capabilities are not part of the Java standard, yet are extremely useful. The JDK class libraries make heavy use of `Unsafe`, especially in places such as the concurrency classes.

However, as the name itself suggests, there are some very powerful and potentially damaging methods contained within `Unsafe`, and it has never been standardized. So, from Java 9 onwards, this class will no longer be available to classes that do not form part of the JDK.

Unfortunately, these features are very widely used by many popular Java frameworks, for performance or flexibility reasons. So even if your Java application doesn't directly call code from `Unsafe`, the chances are that somewhere in your stack, you use a library that does rely on `Unsafe`.

The platform needs to evolve, and the removal of access to internals is a huge step forward for writing maintainable and composable code. However, it's no exaggeration to say that the removal of `Unsafe` has the potential to break every non-trivial Java application currently running.

To most developers, this seems like a backwards incompatible change. From Oracle's point of view, however, the `sun` packages are internal code, and are not guaranteed to remain unchanged. In this view, libraries and frameworks that rely on implementation details rather than public APIs do so at their own risk. This leads to a tension between the needs of the core platform, and the libraries that users rely on.

To resolve this conflict, and given the scope and impact of these changes, the transition must be handled with care and clear communication. Oracle is consulting the wider community and at time of writing a reasonable consensus on how to proceed seems to be emerging.

Change Default Garbage Collector

The current default garbage collector is the parallel collector. The parallel collector is extremely efficient, designed for high-throughput operation and uses very small amounts of CPU time to collect memory. However, the collector must pause the JVM to run a garbage collection cycle (sometimes called a "Stop The World" (STW) operation). These pauses typically last for up to a few hundred milliseconds on heaps of 8 GB or less.

In Java 9, Oracle proposes to change the default collector to the new Garbage First (G1) collector. This uses a more modern GC algorithm that can do some of its work without pausing fully. The aim is to let users set "pause goals" that the JVM will try to adhere to. However, G1 has some drawbacks: it uses much more CPU time overall to collect memory, and still has the possibility of a significant pause. By default, G1 will try to pause for no more than 200ms, unless necessary, which isn't necessarily a huge improvement over parallel.

G1 is also lacking in real-world testing. Despite being available since Java 7, relatively few Java shops have adopted it, so the true impact of changing the default collector is unknown. Applications that run without an explicit choice of collector will be affected by a change of default. Limited research has been done into the percentage of applications that would potentially be affected, but indications are that it could over 50%.

HTTP/2

The HTTP/2 standard is a new version of the Web's primary protocol, HTTP. The previous version, HTTP/1.1, dates from 1999 and has encountered significant problems (such as head-of-line blocking) as the Web has grown. The new standard was created by the Internet Engineering Task Force (IETF) HTTP Working Group, which comprised engineers from major Web companies and browser manufacturers.

NOTE

The basic semantics (including methods) of HTTP have not fundamentally changed in the new standard, but the transport mechanisms are new.

The group summarized some of the key properties of HTTP/2 as follows:

- Same HTTP APIs
- Cheaper requests
- Network- and server-friendliness
- Cache pushing
- Being able to change your mind
- More encryption

The new standard is pragmatic about the way the Web has come to be used; as a general purpose application protocol rather than purely for document retrieval and hypertext transfer. So, for example, in HTTP/2 responses can be interleaved, connections are not closed unless a browser actively navigates away, and HTTP headers are now represented in binary to avoid penalizing small requests and responses (which is the majority of traffic).

In the Java world, HTTP/2 is an opportunity to revisit Java's ancient HTTP API. This dates to Java 1.0 and is designed around a relatively protocol-agnostic framework based on the URL class. This predates the massive dominance of the Web over all other Internet protocols. This API has not kept up with the reality of how the Web is used today.

The new Java API for HTTP/2 is a completely clean sheet, and abandons any pretense of protocol independence. Instead, it's an API purely for HTTP, but is independent of HTTP version. It will provide support for the new framing and connection handling parts of HTTP/2, as well as HTTP/1.1 support for the transitional period.

In the current version of the new API (which may, of course, change before the release of Java 9), a simple HTTP request looks like this:

```
HttpResponse resp = HttpRequest
    .create(new URI("http://www.oreilly.com"))
    .body(noBody())
    .GET().send();
int responseCode = resp.responseCode();
String body = resp.body(asString());

System.out.println(body);
```

This style for the API feels much more modern than the existing legacy HTTP API, and reflects the trend in API design towards fluent (or builder) patterns.

JShell

In many other languages, an interactive environment for exploratory development is provided via a Read-Evaluate-Print-Loop (REPL) tool. In some cases (notably Clojure and other Lisps), the REPL is where developers spend most of their coding time. This is also seen in languages such as Scala or JRuby.

Java previously had the Beanshell scripting language, but it never achieved full standardization, and the project has essentially been abandoned. Java 8 introduced the Nashorn implementation of Javascript on top of the JVM, and included the `jjs` REPL. Due to Nashorn's tight integration with Java, this could be a useful environment for playing with Java in an interactive manner. However, it still wasn't Java.

As part of the development of Java 9, Project Kulla was started, to look at producing a Java REPL that would provide as close an experience to "full Java" as possible. The project had some strict goals, such as not to introduce new non-Java syntax. Instead, it disables some features of the language that are not useful for interactive development in order to provide a less awkward working environment.

In JShell, statements and expressions are evaluated immediately in the context of an execution state. This means that they do not have to be packaged into classes, and methods can also be free-standing. JShell uses "snippets" of code to provide this top-level execution environment.

In the environment, expressions can be freely entered and JShell will automatically create temporary variables to hold the resulting values and keep them in scope for later use:

```
-> 3 * (4 + 7)
|  Expression value is: 33
|    assigned to temporary variable $1 of type int

-> System.out.println($1);
33
```

New classes can easily be defined:

```
-> class Pet {}
|  Added class Pet

-> class Dog extends Pet {}
|  Added class Dog
```

JShell also has commands, which all start with / to access REPL features. For example:

```
-> /help
Type a Java language expression, statement, or declaration.
Or type one of the following commands:

/l  or /list [all]        -- list the source you have typed

[additional output]

/?  or /help              -- this help message
       /!                 -- re-run last snippet
       /<n>               -- re-run n-th snippet
       /-<n>              -- re-run n-th previous snippet

Supported shortcuts include:
<tab>       -- show possible completions for the current text
```

Just like REPL environments in other languages, JShell lets you use the REPL to demonstrate Java language features very simply and quickly. In turn, this makes JShell a great learning tool, similar in experience to Scala's REPL.

Further Out

Oracle does not release firm plans more than one release ahead, relying instead on a roadmap of features for future releases. As a result, the features and possible developments discussed in this section cannot be definitively tied to any specific release.

Project Panama

Oracle has already announced Project Panama, a new effort to define a Foreign Function Interface (FFI) for the JVM. The name evokes the Panama canal, an infrastructure project designed to link the Pacific to the Atlantic. Similarly, Project Panama is about bridging between the managed world of Java and the unmanaged world of C and other runtimes.

> If non-Java programmers find some library useful and easy to access, it should be similarly accessible to Java programmers.
>
> —John Rose

The ultimate goal is to be able to directly bind native functions (such as the contents of shared libraries or operating-system calls) to Java methods. This has always been possible using Java's Java Native Interface (JNI), but the interface is inconvenient and rather limited. This has led to a significant barrier to entry for mixing native code into a Java project.

Project Panama has a difficult task ahead of it, not least because Java's culture has always been about safe programming, as a departure from the pitfalls found in languages such as C and C++. To evolve Java's access to native code without sacrificing that safety is a major undertaking, but would be of huge benefit to millions of Java developers worldwide.

Project Valhalla

Another area of major work beyond Java 9 is Project Valhalla. This is an experimental project focused on new features for the Java language. Currently, the features that are under discussion are enhanced generics and value types.

Enhanced generics are a proposed feature that would let Java developers write code that uses primitive types as type parameters for collections, such as `List<int>`. This is problematic in the current language and JVM because there is no type in Java that is a supertype of both Object and int. That is, Java's type system does not have a single root.

Currently, the prototyping uses an approach called "any" type variables, to mean that the type variable can range over both reference types and primitives. However, this design contains some subtleties that have to be approached carefully. For example, `List<int>` and `List<String>` could not have a supertype more specific than Object in Java's existing type system.

One possibility is that `List<Integer>` and `List<String>` could continue to be represented at runtime by `List.class`, but with `List<int>` being represented by a different runtime type and class file.

The Internet of Things

Software is not a static field, and new areas of interest continue to emerge. One of the most eagerly anticipated and hyped is the so-called Internet of Things (IoT). This is the idea that devices with very limited compute capability compared to a laptop or phone will nevertheless become Internet-enabled and able to provide useful and valuable data streams to their owners.

> Java has inspired a lot of hatred, but it's been incredibly influential in building modern enterprise software, along with the tools we use to develop, maintain, and deploy that software.
>
> —Mike Loukides

Over the years, a lot of the criticism (both justified and not) that has been flung in Java's direction has abated, replaced by something closer to grudging, involuntary respect.

It is therefore not surprising that, given Java's influence in the enterprise, application teams working towards IoT have developed stacks that leverage Java's strengths and robustness for use with a world of devices possessed of limited capability.

It's still unclear whether the much-discussed revolution of IoT will actually take place. While the raw technology is now in place, major issues such as security, bandwidth, and data handling remain. For that matter, the industry has yet to decide whether a device's "owner" and beneficiary of the device's data value is the purchaser or the supplier.

In any event, if the IoT does become mainstream, then Java is extremely well-placed to become a major part of the architecture of the systems that will be needed to deliver it.

Conclusion

The road from Java's first public alpha of 1.0 to today has been long and full of technical advances and interesting adventures. Along the way, Java has flourished, and has become one of the world's most important and widely-used programming environments.

How long will Java continue to be as ubiquitous as it is today? No-one knows, but the ecosystem today is flourishing and the immediate course that has been set seems fair. Which means, of course, that it's time to raise a toast and wish Java a very Happy Birthday.

About the Author

Ben Evans is the Cofounder and Technology Fellow of jClarity, a startup that delivers performance tools for development and ops teams. He helps to organize the London Java Community and represents them on the Java Community Process Executive Committee, where he works to define new standards for the Java ecosystem. He is a Java Champion; JavaOne Rockstar; coauthor of *The Well-Grounded Java Developer* and *Java in a Nutshell 6E*. He lives in London, but is usually found traveling the world consulting, speaking, and educating on the Java platform, performance analysis, system architecture, and related topics.