Chapter 1. Introduction to the Java Environment

Welcome to Java 11.

That version number probably surprises you as much as it does us. It seems like only yesterday that Java 5 was the new thing, and yet here we are, 14 years and 6 major versions later.

You may be coming to the Java ecosystem from another language, or maybe this is your first programming language. Whatever road you may have traveled to get here, welcome—we're glad you've arrived.

Java is a powerful, general-purpose programming environment. It is one of the most widely used programming languages in the world, and has been exceptionally successful in business and enterprise computing.

In this chapter, we'll set the scene by describing the Java language (which programmers write their applications in), the Java Virtual Machine (which executes those applications), and the Java ecosystem (which provides a lot of the value of the programming environment to development teams).

We'll briefly cover the history of the Java language and virtual machine, before moving on to discuss the lifecycle of a Java program and clear up some common questions about the differences between Java and other environments.

At the end of the chapter, we'll introduce Java security, and discuss some of the aspects of Java that relate to secure coding.

The Language, the JVM, and the Ecosystem

The Java programming environment has been around since the late 1990s. It comprises the Java language, and the supporting runtime, otherwise known as the Java Virtual Machine (JVM).

At the time that Java was initially developed, this split was considered novel, but recent trends in software development have made it more commonplace. Notably, Microsoft's .NET environment, announced a few years after Java, adopted a very similar approach to platform architecture.

One important difference between Microsoft's .NET platform and Java is that Java was always conceived as a relatively open ecosystem of multiple vendors, albeit led by a steward who owns the technology. Throughout Java's history, these vendors have both cooperated and competed on aspects of Java technology.

One of the main reasons for the success of Java is that this ecosystem is a standardized environment. This means there are specifications for the technologies that comprise the environment. These standards give the developer and consumer confidence that the technology will be compatible with other components, even if they come from a different technology vendor.

The current steward of Java is Oracle Corporation (who acquired Sun Microsystems, the originator of Java). Other corporations, such as Red Hat, IBM, Amazon, AliBaba, SAP, Azul Systems, and Fujitsu are also heavily involved in producing implementations of standardized Java technologies.

TIP

As of Java 11, the primary reference implementation of Java is the open source OpenJDK, which many of these companies collaborate on and base their shipping products upon.

Java actually comprises several different but related environments and specifications, such as Java Mobile Edition (Java ME),[1] Java Standard Edition (Java SE), and Java Enterprise Edition (Java EE).[2] In this book, we'll only cover Java SE, version 11, with some historical notes related to when certain features were introduced into the platform.

We will have more to say about standardization later, so let's move on to discuss the Java language and JVM as separate but related concepts.

What Is the Java Language?

Java programs are written as source code in the Java language. This is a human-readable programming language, which is strictly class based and object oriented. The language syntax is deliberately modeled on that of C and C++ and it was explicitly intended to be familiar to programmers coming from those languages.

NOTE

Although the source code is similar to C++, in practice Java includes features and a managed runtime that has more in common with more dynamic languages such as Smalltalk.

Java is considered to be relatively easy to read and write (if occasionally a bit verbose). It has a rigid grammar and simple program structure, and is intended to be easy to learn and to teach. It builds on industry experience with languages like C++ and tries to remove complex features as well as preserving "what works" from previous programming languages.

Overall, Java is intended to provide a stable, solid base for companies to develop business-critical applications. As a programming language, it has a relatively conservative design and a slow rate of change. These properties are a conscious attempt to serve the goal of protecting the investment that organizations have made in Java technology.

The language has undergone gradual revision (but no complete rewrites) since its inception in 1996. This does mean that some of Java's original design choices, which were expedient in the late 1990s, are still affecting the language today—see Chapters 2 and 3 for more details.

Java 8 added the most radical changes seen in the language for almost a decade (some would say since the birth of Java). Features like lambda expressions and the overhaul of the core Collections code were enormously popular and changed forever the way that Java developers write code. Since then, the platform has produced a release (Java 9) that adds a major (and long-delayed) feature: the platform modules system (JPMS).

With that release, the project has transitioned to a new, much faster release model where new Java versions are released every six months—bringing us up to Java 11. The Java language is governed by the Java

Language Specification (JLS), which defines how a conforming implementation must behave.

What Is the JVM?

The JVM is a program that provides the runtime environment necessary for Java programs to execute. Java programs cannot run unless there is a JVM available for the appropriate hardware and OS platform we wish to execute on.

Fortunately, the JVM has been ported to run on a large number of environments—anything from a set-top box or Blu-ray player to a huge mainframe will probably have a JVM available for it.

Java programs are typically started from a command line like this:

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

This brings up the JVM as an operating system process that provides the Java runtime environment, and then executes our program in the context of the freshly started (and empty) virtual machine.

It is important to understand that when the JVM takes in a Java program for execution, the program is not provided as Java language source code. Instead, the Java language source must have been converted (or compiled) into a form known as Java bytecode. Java bytecode must be supplied to the JVM in a format called class files (which always have a *.class* extension).

The JVM provides an *execution environment* for the program. It starts an interpreter for the bytecode form of the program that steps through one bytecode instruction at a time. However, production JVMs also provide a runtime compiler that will accelerate the important parts of the program by replacing them with equivalent compiled machine code.

You should also be aware that both the JVM and the user program are capable of spawning additional threads of execution, so that a user program may have many different functions running simultaneously.

The design of the JVM built on many years of experience with earlier programming environments, notably C and C++, so we can think of it as

having several different goals—which are all intended to make life easier for the programmer:

- Comprise a container for application code to run inside
- Provide a secure and reliable execution environment as compared to C/C++
- Take memory management out of the hands of developers
- Provide a cross-platform execution environment

These objectives are often mentioned together in discussions of the platform.

We've already mentioned the first of these goals, when we discussed the JVM and its bytecode interpreter—it functions as the container for application code.

We'll discuss the second and third goals in Chapter 6, when we talk about how the Java environment deals with memory management.

The fourth goal, sometimes called "write once, run anywhere" (WORA), is the property that Java class files can be moved from one execution platform to another, and they will run unaltered provided a JVM is available.

This means that a Java program can be developed (and converted to class files) on a machine running macOS, and then the class files can be moved to Linux or Microsoft Windows (or other platforms) and the Java program will run without any further work needed.

NOTE

The Java environment has been very widely ported, including to platforms that are very different from mainstream platforms like Linux, macOS, and Windows. In this book, we use the phrase "most implementations" to indicate those platforms that the majority of developers are likely to encounter; macOS, Windows, Linux, BSD Unix, and the like are all considered "mainstream platforms" and count within "most implementations."

In addition to these four primary goals, there is another aspect of the JVM's design that is not always recognized or discussed—it makes use of runtime information to self-manage.

Software research in the 1970s and 1980s revealed that the runtime behavior of programs has a large amount of interesting and useful patterns that cannot be deduced at compile time. The JVM was the first truly mainstream platform to make use of this research.

It collects runtime information to make better decisions about how to execute code. That means that the JVM can monitor and optimize a program running on it in a manner not possible for platforms without this capability.

A key example is the runtime fact that not all parts of a Java program are equally likely to be called during the lifetime of the program—some portions will be called far, far more often than others. The Java platform takes advantage of this fact with a technology called just-in-time (JIT) compilation.

In the HotSpot JVM (which was the JVM that Sun first shipped as part of Java 1.3, and is still in use today), the JVM first identifies which parts of the program are called most often—the "hot methods." Then, the JVM compiles these hot methods directly into machine code, bypassing the JVM interpreter.

The JVM uses the available runtime information to deliver higher performance than was possible from purely interpreted execution. In fact, the optimizations that the JVM uses now in many cases produce performance that surpasses compiled C and C++ code.

The standard that describes how a properly functioning JVM must behave is called the JVM Specification.

What Is the Java Ecosystem?

The Java language is easy to learn and contains relatively few abstractions, compared to other programming languages. The JVM provides a solid, portable, high-performance base for Java (or other languages) to execute on. Taken together, these two connected

technologies provide a foundation that businesses can feel confident about when choosing where to base their development efforts.

The benefits of Java do not end there, however. Since Java's inception, an extremely large ecosystem of third-party libraries and components has grown up. This means that a development team can benefit hugely from the existence of connectors and drivers for practically every technology imaginable—both proprietary and open source.

In the modern technology ecosystem it is now rare indeed to find a technology component that does *not* offer a Java connector. From traditional relational databases, to NoSQL, to every type of enterprise monitoring system, to messaging systems, to Internet of Things (IoT)—everything integrates with Java.

It is this fact that has been a major driver of adoption of Java technologies by enterprises and larger companies. Development teams have been able to unlock their potential by making use of preexisting libraries and components. This has promoted developer choice and encouraged open, best-of-breed architectures with Java technology cores.

NOTE

Google's Android environment is sometimes thought of as being "based on Java." However, the picture is actually more complicated. Android code is written in Java but originally used a different implementation of Java's class libraries along with a cross compiler to convert to a different file format for a non-Java virtual machine.

The combination of a rich ecosystem and a first-rate virtual machine with an open standard for program binaries makes the Java platform a very attractive execution target. In fact, there are a large number of non-Java languages that target the JVM and also interoperate with Java (which allows them to piggy-back off the platform's success). These languages include Kotlin, Scala, Groovy, and many others. While all of them are small compared to Java, they have distinct niches within the Java world, and provide a source of innovation and healthy competition to Java.

A Brief History of Java and the JVM

Java 1.0 (1996)

> This was the first public version of Java. It contained just 212 classes organized in eight packages. The Java platform has always had an emphasis on backward compatibility, and code written with Java 1.0 will still run today on Java 11 without modification or recompilation.

Java 1.1 (1997)

> This release of Java more than doubled the size of the Java platform. This release introduced “inner classes” and the first version of the Reflection API.

Java 1.2 (1998)

> This was a very significant release of Java; it tripled the size of the Java platform. This release marked the first appearance of the Java Collections API (with sets, maps, and lists). The many new features in the 1.2 release led Sun to rebrand the platform as “the Java 2 Platform.” The term “Java 2” was simply a trademark, however, and not an actual version number for the release.

Java 1.3 (2000)

> This was primarily a maintenance release, focused on bug fixes, stability, and performance improvements. This release also brought in the HotSpot Java Virtual Machine, which is still in use today (although heavily modified and improved since then).

Java 1.4 (2002)

> This was another fairly big release, adding important new functionality such as a higher-performance, low-level I/O API; regular expressions for text handling; XML and XSLT libraries; SSL support; a logging API; and cryptography support.

Java 5 (2004)

> This large release of Java introduced a number of changes to the core language itself including generic types, enumerated types (enums), annotations, varargs methods, autoboxing, and a new `for` loop. These changes were considered significant enough to change the major version number, and to start numbering as major releases. This release included 3,562 classes and interfaces in 166 packages. Notable additions included utilities for concurrent

programming, a remote management framework, and classes for the remote management and instrumentation of the Java VM itself.

Java 6 (2006)

This release was also largely a maintenance and performance release. It introduced the Compiler API, expanded the usage and scope of annotations, and provided bindings to allow scripting languages to interoperate with Java. There were also a large number of internal bug fixes and improvements to the JVM and the Swing GUI technology.

Java 7 (2011)

The first release of Java under Oracle's stewardship included a number of major upgrades to the language and platform. The introduction of `try`-with-resources and the NIO.2 API enabled developers to write much safer and less error-prone code for handling resources and I/O. The Method Handles API provided a simpler and safer alternative to reflection; in addition, it opened the door for `invokedynamic` (the first new bytecode since version 1.0 of Java).

Java 8 (2014) (LTS)

This was a huge release—potentially the most significant changes to the language since Java 5 (or possibly ever). The introduction of lambda expressions provided the ability to significantly enhance the productivity of developers, the Collections were updated to make use of lambdas, and the machinery required to achieve this marked a fundamental change in Java's approach to object orientation. Other major updates include a new date and time API, and major updates to the concurrency libraries.

Java 9 (2017)

Significantly delayed, this release introduced the new platform modularity feature, which allows Java applications to be packaged into deployment units and modularize the platform runtime. Other changes include a new default garbage collection algorithm, a new API for handling processes, and some changes to the way that frameworks can access the internals.

Java 10 (March 2018)

This marks the first release under the new release cycle. This release contained a relatively small amount of new features (due to its six-month development lifetime). New syntax for type inference

was introduced, along with some internal changes (including GC tweaks and an experimental new compiler).

Java 11 (September 2018) (LTS)

The current version, also developed over a short six-month window, this release is the first modular Java to be considered as a long-term support (LTS) release. It adds relatively few new features that are directly visible to the developer—primarily Flight Recorder and the new HTTP/2 API. There are some additional internal changes, but this release is primarily for stabilization.

As it stands, the only current production versions are Java 8 and 11—the LTS releases. Due to the highly significant changes that are introduced by modules, Java 8 has been grandfathered in as an LTS release to provide extra time for teams and applications to migrate to a supported modular Java.

The Lifecycle of a Java Program

To better understand how Java code is compiled and executed, and the difference between Java and other types of programming environments, consider the pipeline in Figure 1-1.

Figure 1-1. How Java code is compiled and loaded

This starts wth Java source, and passes it through the `javac` program to produce class files—which contain the source code compiled to Java bytecode. The class file is the smallest unit of functionality the platform will deal with, and the only way to get new code into a running program.

New class files are onboarded via the classloading mechanism (see Chapter 10 for a lot more detail on how classloading works). This makes the new type available to the interpreter for execution.

Frequently Asked Questions

In this section, we'll discuss some of the most frequently asked questions about Java and the lifecycle of programs written in the Java environment.

WHAT IS BYTECODE?

When developers are first introduced to the JVM, they sometimes think of it as “a computer inside a computer.” It’s then easy to imagine bytecode as “machine code for the CPU of the internal computer” or “machine code for a made-up processor.”

In fact, bytecode is not actually very similar to machine code that would run on a real hardware processor. Instead, computer scientists would call bytecode a type of *intermediate representation*—a halfway house between source code and machine code.

The whole aim of bytecode is to be a format that can be executed efficiently by the JVM’s interpreter.

IS JAVAC A COMPILER?

Compilers usually produce machine code, but `javac` produces bytecode, which is not that similar to machine code. However, class files are a bit like object files (like Windows *.dll* files, or Unix *.so* files)—and they are certainly not human readable.

In theoretical computer science terms, `javac` is most similar to the *front half* of a compiler—it creates the intermediate representation that can then be used later to produce (emit) machine code.

However, because creation of class files is a separate build-time step that resembles compilation in C/C++, many developers consider running `javac` to be compilation. In this book, we will use the terms “source code compiler” or "`javac` compiler” to mean the production of class files by `javac`.

We will reserve “compilation” as a standalone term to mean JIT compilation—as it’s JIT compilation that actually produces machine code.

WHY IS IT CALLED “BYTECODE”?

The instruction code (opcode) is just a single byte (some operations also have parameters that follow them in the bytestream), so there are only 256 possible instructions. In practice, some are unused—about 200 are in use, but some of them aren’t emitted by recent versions of `javac`.

IS BYTECODE OPTIMIZED?

In the early days of the platform, `javac` produced heavily optimized bytecode. This turned out to be a mistake. With the advent of JIT compilation, the important methods are going to be compiled to very fast machine code. It's therefore very important to make the job of the JIT compiler easier—as there are much bigger gains available from JIT compilation than there are from optimizing bytecode, which will still have to be interpreted.

IS BYTECODE REALLY MACHINE INDEPENDENT? WHAT ABOUT THINGS LIKE ENDIANNESS?

The format of bytecode is always the same, regardless of what type of machine it was created on. This includes the byte ordering (sometimes called "endianness") of the machine. For readers who are interested in the details, bytecode is always big-endian.

IS JAVA AN INTERPRETED LANGUAGE?

The JVM is basically an interpreter (with JIT compilation to give it a big performance boost). However, most interpreted languages (such as PHP, Perl, Ruby, and Python) directly interpret programs from source form (usually by constructing an abstract syntax tree from the input source file). The JVM interpreter, on the other hand, requires class files—which, of course, require a separate source code compilation step with `javac`.

CAN OTHER LANGUAGES RUN ON THE JVM?

Yes. The JVM can run any valid class file, so this means that non-Java languages can run on the JVM in one of two ways. First, they could have a source code compiler (similar to `javac`) that produces class files, which would run on the JVM just like Java code (this is the approach taken by languages like Scala).

Alternatively, a non-Java language could implement an interpreter and runtime in Java, and then interpret the source form of their language directly. This second option is the approach taken by languages like JRuby (but JRuby has a very sophisticated runtime that is capable of *secondary JIT compilation* in some circumstances).

Java Security

Java has been designed from the ground up with security in mind; this gives it a great advantage over many other existing systems and platforms. The Java security architecture was designed by security experts and has been studied and probed by many other security experts since the inception of the platform. The consensus is that the architecture itself is strong and robust, without any security holes in the design (at least none that have been discovered yet).

Fundamental to the design of the security model is that bytecode is heavily restricted in what it can express—there is no way, for example, to directly address memory. This cuts out entire classes of security problems that have plagued languages like C and C++. Furthermore, the VM goes through a process known as *bytecode verification* whenever it loads an untrusted class, which removes a further large class of problems (see Chapter 10 for more about bytecode verification).

Despite all this, however, no system can guarantee 100% security, and Java is no exception.

While the design is still theoretically robust, the implementation of the security architecture is another matter, and there is a long history of security flaws being found and patched in particular implementations of Java.

In particular, the release of Java 8 was delayed, at least partly, due to the discovery of a number of security problems that required considerable effort to fix.

In all likelihood, security flaws will continue to be discovered (and patched) in Java VM implementations. For practical server-side coding, Java remains perhaps the most secure general-purpose platform currently available, especially when kept patched up to date.

Comparing Java to Other Languages

In this section, we'll briefly highlight some differences between the Java platform and other programming environments you may be familiar with.

Java Compared to C

- Java is object oriented; C is procedural.
- Java is portable as class files; C needs to be recompiled.
- Java provides extensive instrumentation as part of the runtime.
- Java has no pointers and no equivalent of pointer arithmetic.
- Java provides automatic memory management via garbage collection.
- Java has no ability to lay out memory at a low level (no structs).
- Java has no preprocessor.

Java Compared to C++

- Java has a simplified object model compared to C++.
- Java's dispatch is virtual by default.
- Java is always pass-by-value (but one of the possibilities for Java's values is object references).
- Java does not support full multiple inheritance.
- Java's generics are less powerful (but also less dangerous) than C++ templates.
- Java has no operator overloading.

Java Compared to Python

- Java is statically typed; Python is dynamically typed.
- Java is multithreaded; Python only allows one thread to execute Python at once.
- Java has a JIT; the main implementation of Python does not.
- Java's bytecode has extensive static checks; Python's bytecode does not.

Java Compared to JavaScript

- Java is statically typed; JavaScript is dynamically typed.
- Java uses class-based objects; JavaScript is prototype based.
- Java provides good object encapsulation; JavaScript does not.
- Java has namespaces; JavaScript does not.
- Java is multithreaded; JavaScript is not.

Answering Some Criticisms of Java

Java has had a long history in the public eye and, as a result, has attracted its fair share of criticism over the years. Some of this negative press can be attributed to some technical shortcomings combined with rather overzealous marketing in the first versions of Java.

Some criticisms have, however, entered technical folklore despite no longer being very accurate. In this section, we'll look at some common grumbles and the extent to which they're true for modern versions of the platform.

Overly Verbose

The Java core language has sometimes been criticized as overly verbose. Even simple Java statements such as `Object o = new Object();` seem to be repetitious—the type `Object` appears on both the left and right side of the assignment. Critics point out that this is essentially redundant, that other languages do not need this duplication of type information, and that many languages support features (e.g., type inference) that remove it.

The counterpoint to this argument is that Java was designed from the start to be easy to read (code is read more often than written) and that many programmers, especially novices, find the extra type information helpful when reading code.

Java is widely used in enterprise environments, which often have separate dev and ops teams. The extra verbosity can often be a blessing when you are responding to an outage call, or when you need to

maintain and patch code that was written by developers who have long since moved on.

In recent versions of Java, the language designers have attempted to respond to some of these points, by finding places where the syntax can become less verbose and by making better use of type information. For example:

```
// Files helper methods
byte[] contents =
   Files.readAllBytes(Paths.get("/home/ben/myFile.bin"));

// Diamond syntax for repeated type information
List<String> l = new ArrayList<>();

// Local variables can be type inferred
var threadPool = Executors.newScheduledThreadPool(2);
// Lambda expressions simplify Runnables
threadPool.submit(() -> { System.out.println("On Threadpool"); });
```

However, Java's overall philosophy is to make changes to the language only very slowly and carefully, so the pace of these changes may not satisfy detractors completely.

Slow to Change

The original Java language is now well over 20 years old, and has not undergone a complete revision in that time. Many other languages (e.g., Microsoft's C#) have released backward-incompatible versions in the same period, and some developers criticize Java for not doing likewise.

Furthermore, in recent years, the Java language has come under fire for being slow to adopt language features that are now commonplace in other languages.

The conservative approach to language design that Sun (and now Oracle) has taken is an attempt to avoid imposing the costs and externalities of misfeatures on a very large user base. Many Java shops have made major investments in the technology, and the language designers have taken seriously the responsibility of not disrupting the existing user and install base.

Each new language feature needs to be very carefully thought about—not only in isolation, but in terms of how it will interact with all the existing features of the language. New features can sometimes have

impacts beyond their immediate scope—and Java is widely used in very large codebases, where there are more potential places for an unexpected interaction to manifest.

It is almost impossible to remove a feature that turns out to be incorrect after it has shipped. Java has a couple of misfeatures (such as the finalization mechanism) that it has never been possible to remove safely without impacting the install base. The language designers have taken the view that extreme caution is required when evolving the language.

Having said that, the new language features that have arrived in recent versions are a significant step toward addressing the most common complaints about missing features, and should cover many of the idioms that developers have been asking for.

Performance Problems

The Java platform is still sometimes criticized as being slow—but of all the criticisms that are leveled at the platform, this is probably the one that is least justified. It is a genuine myth about the platform.

Release 1.3 of Java brought in the HotSpot Virtual Machine and its JIT compiler. Since then, there has been over 15 years of continual innovation and improvement in the virtual machine and its performance. The Java platform is now blazingly fast, regularly winning performance benchmarks on popular frameworks, and even beating native-compiled C and C++.

Criticism in this area appears to be largely caused by a folk memory that Java was slow at some point in the past. Some of the larger and more sprawling architectures that Java has been used within may also have contributed to this impression.

The truth is that any large architecture will require benchmarking, analysis, and performance tuning to get the best out of it—and Java is no exception.

The core of the platform—language and JVM—was and remains one of the fastest general-use environments available to the developer.

Insecure

During 2013 there were a number of security vulnerabilities in the Java platform, which caused the release date of Java 8 to be pushed back. Even before this, some people had criticized Java's record of security vulnerabilities.

Many of these vulnerabilities involved the desktop and GUI components of the Java system, and wouldn't affect websites or other server-side code written in Java.

All programming platforms have security issues at times, and many other languages have a comparable history of security vulnerabilities that have been significantly less well publicized.

Too Corporate

Java is a platform that is extensively used by corporate and enterprise developers. The perception that it is too corporate is therefore an unsurprising one—Java has often been perceived as lacking the "free-wheeling" style of languages that are deemed to be more community oriented.

In truth, Java has always been, and remains, a very widely used language for community and free or open source software development. It is one of the most popular languages for projects hosted on GitHub and other project hosting sites.

Finally, the most widely used implementation of the language itself is based on OpenJDK—which is itself an open source project with a vibrant and growing community.