

Well-Grounded Java Developer

SECOND EDITION

Martijn Verburg

Jason Clark

Benjamin Evans



MANNING



MEAP Edition
Manning Early Access Program
The Well-Grounded Java Developer
Second Edition
Version 6

Copyright 2021 Manning Publications

For more information on this and other Manning titles go to
manning.com

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javalib>

welcome

Thanks for purchasing the MEAP for *The Well-Grounded Java Developer, Second Edition*.

Although Java is 25 years old, its design goals and history mean valuable new functionality continues to appear release after release. It's vibrant, broadly supported, and a high performance basis for vast amounts of software in the wild. Recent changes to the release cycle mean if anything we're seeing new functionality show up more often - not what you might expect for language after a quarter of a century!

This book is written for Java developers looking to catch up on those latest techniques and additions. If you're using Java 8 or earlier, you'll see the exciting new capabilities in upgrading to Java 11, along with plenty of reasons to convince your team to upgrade. We'll also peek at what the future holds for the Java platform in coming releases.

Being well-grounded also means knowing the fundamentals of your platform. If you've ever wanted to understand better what's in a class file, how classloaders work, or what the key performance tuning options are, we've got you covered. We'll dig deep on concurrency as well to see how to harness all those cores in your modern computer. One of the best ways to understand programming more deeply is to learn another language.

In the past decade support for other languages on the JVM has come a long way, and we'll introduce two that we feel are worth your time (Kotlin and Clojure). While these languages might not replace Java for you, both bring compelling features and will stretch your concepts of what can be accomplished on the JVM.

Whether you're just looking for what's new in Java, or you're looking to boost your skills to the next level of mastery, we hope that *The Well-Grounded Java Developer* helps you learn and grow.

Don't be shy to post questions, and feedback in the [liveBook's Discussion Forum](#).

—Benjamin J Evans, Jason R Clark, and Martijn Verburg

brief contents

PART 1: FROM 8 TO 11 TO 17

- 1 Introducing modern Java*
- 2 Java modules*
- 3 Java 17*

PART 2: UNDER THE HOOD

- 4 Class files and bytecode*
- 5 Java concurrency fundamentals*
- 6 JDK concurrency libraries*
- 7 Understanding Java performance*

PART 3: NON-JAVA LANGUAGES ON THE JVM

- 8 Alternative JVM languages*
- 9 Kotlin*
- 10 Clojure: a different view of programming*

PART 4: BUILD AND DEPLOYMENT

- 11 Building with Gradle & Maven*
- 12 Running Java in containers*
- 13 Testing beyond JUnit*

PART 5: JAVA FRONTIERS

- 14 Advanced functional programming*
- 15 Advanced concurrent programming*
- 16 Modern internals*
- 17 Future Java*

APPENDIXES

- A Installing Java 11 & Builds and licenses*
- B Review of Streams and functional programming in Java*

Introducing modern Java



This chapter covers

- Java as a platform and a language
- The new Java release model
- Enhanced Type inference (`var`)
- Incubating and Preview Features
- Changing the language
- Small language changes in Java 11

Welcome to Java at the end of 2021. It is an exciting time. Java 17, the latest Long-Term Support (LTS) release shipped - in September 2021, and the first and most adventurous teams are starting to move to it.

At the time of writing, apart from a few trailblazers, Java applications are more-or-less evenly split between running on Java 11 (released September 2018) and the much older Java 8 (2014). There is a lot to recommend Java 11, especially for teams that are deploying in the cloud, but some developers have been a little slow to adopt it.

So, in the first part of this book, we are going to spend some time introducing some new features that have arrived in Java 11 and 17. Hopefully, this discussion will help convince some teams and managers who may be reluctant to upgrade from 8 that things are better than ever in the newer versions.

Our focus for this chapter is going to be Java 11 because a) it's the LTS version with the largest market share and b) there has not been any noticeable adoption of Java 17 yet. However, in Chapter 3 we will introduce the new features of Java 17 to bring you all the way up to date.

Let's get underway by discussing the language versus platform duality that lies at the heart of

modern Java. This is a critically important point that we'll come back to several times throughout the book, so it's an essential one to grasp right at the start.

1.1 The language and the platform

"Java" as a term can refer to one of several related concepts. In particular, it could mean either the human-readable programming language or the much broader "Java platform".

Surprisingly, different authors sometimes give slightly different definitions of what constitutes the language and platform. This can lead to a lack of clarity and some confusion about the differences between the two and about which provides the various programming features that application code uses.

Let's make that distinction clear right now, as it cuts to the heart of a lot of the topics in this book. Here are our definitions:

- *The Java language* — The Java language is the statically typed, object-oriented language that we lightly lampooned in the “About This Book” section. Hopefully, it's already very familiar to you. One obvious point about source code written in the Java language is that it's human-readable (or it should be!).
- *The Java platform* — The platform is the software that provides a runtime environment. It's the JVM that links and executes your code as provided to it in the form of (not human-readable) class files. It doesn't directly interpret Java language source files, but instead requires them to be converted to class files first.

One of the big reasons for the success of Java as a software system is that it's a standard. This means that it has specifications that describe how it's supposed to work. Standardization allows different vendors and project groups to produce implementations that should all, in theory, work the same way. The specs don't make guarantees about how well different implementations will perform when handling the same task, but they can provide assurances about the correctness of the results.

There are several separate specs that govern the Java system—the most important are the Java Language Specification (JLS) and the JVM Specification (VMSpec). This separation is taken very seriously in modern Java; in fact, the VMSpec no longer makes any reference whatsoever to the JLS directly. We'll have a bit more to say about the differences between these two specs later in the book.

One obvious question, when you're faced with the described duality, is, “What's the link between them?” If they're now separate, then how do they come together to make the Java system?

The link between the language and platform is the shared definition of the class file format (the .class files). A serious study of the class file definition will reward you, and it's one of the ways a good Java programmer can start to become a great one. In figure 1.1, you can see the full process by which Java code is produced and used.

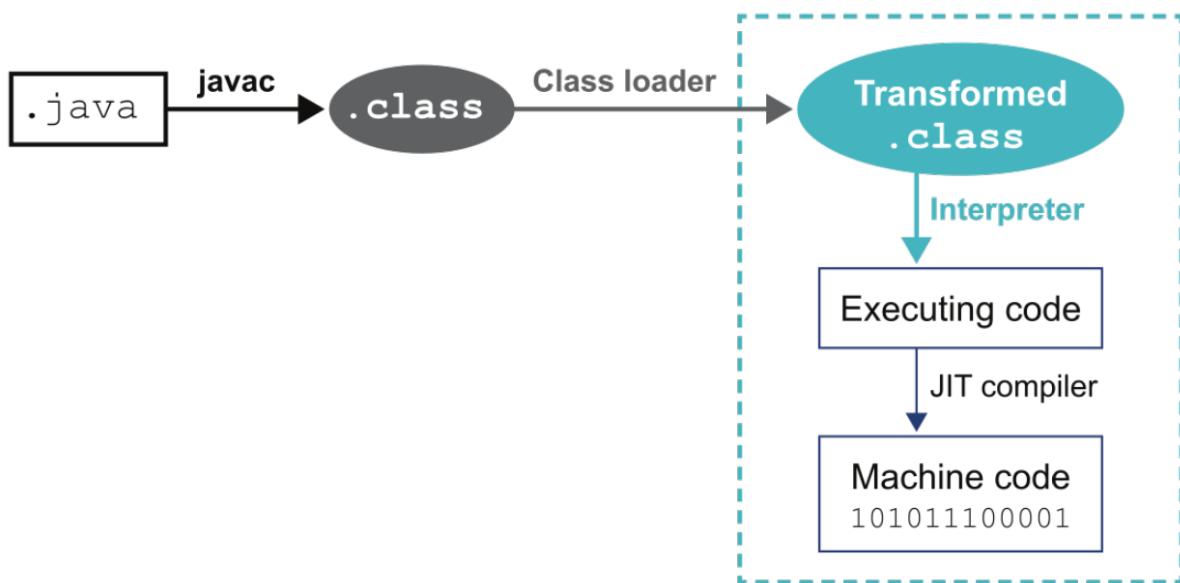


Figure 1.1 Java source code is transformed into .class files, then manipulated at load time before being JIT-compiled.

As you can see in the figure, Java code starts life as human-readable Java source, and it's then compiled by `javac` into a .class file. This is then loaded into a JVM. It's prevalent for classes to be manipulated and altered during the loading process. Many of the most popular Java frameworks will transform classes as they're loaded to inject dynamic behavior such as instrumentation or alternative lookups for classes to load.

NOTE Classloading is an essential feature of the Java platform, and we will hear a lot more about it in Chapter 4.

Is Java a compiled or interpreted language?

The standard picture of Java is of a language that's compiled into .class files before being run on a JVM. If pressed, many developers can also explain that bytecode starts off by being interpreted by the JVM but will undergo just-in-time (JIT) compilation at some later point. Here, however, many people's understanding breaks down in a somewhat hazy conception of bytecode as basically being "machine code for an imaginary or simplified CPU".

In fact, JVM bytecode is more like a halfway house between human-readable source and machine code. In the technical terms of compiler theory, bytecode is really a form of intermediate language (IL) rather than actual machine code. This means that the process of turning Java source into bytecode isn't really compilation in the sense that a C++ or a Go

programmer would understand it, and javac isn't a compiler in the same sense as gcc is—it's really a class file generator for Java source code. The real compiler in the Java ecosystem is the JIT compiler, as you can see in figure 1.1.

Some people describe the Java system as “dynamically compiled.” This emphasizes that the compilation that matters is the JIT compilation at runtime, not the creation of the class file during the build process.

NOTE **The existence of the source code compiler, `javac` leads many developers to think of Java as a static, compiled language. One of the big secrets is that Java is actually a very dynamic environment - it's just hidden a bit below the surface.**

So, the real answer to, “Is Java compiled or interpreted?” is “Both.”

With the distinction between language and platform now clearer, let’s move on to talk about the new Java release model.

1.2 The new Java release model

Java was not always an open source language, but following an announcement at the JavaOne conference in 2006, the source code for Java itself (minus a few bits that Sun didn’t own the source for) was released under the [GPLv2+CE](#) license.

This was around the time of the release of Java 6, so Java 7 was the first version of Java to be developed under an open source software (OSS) license. The primary focus for open source development of the Java platform since then has been the [OpenJDK](#) project, and that continues to this day.

A lot of the project discussion takes place on mailing lists that discuss aspects of the overall codebase. There are 'permanent' lists such as core-libs (core libraries), as well as more transient lists that are formed as part of specific OpenJDK projects such as lambda-dev (lambdas), which then become inactive when a particular project has been completed.

In general, these lists have been the relevant forums for discussing possible future features, allowing developers from the wider community to participate in the process of producing new versions of Java.

NOTE **Sun Microsystems was acquired by Oracle shortly before Java 7 was released. Therefore, all of Oracle’s releases of Java have been based on the open-source codebase.**

The open-source releases of Java had settled into a feature-driven release cycle, where a single marquee feature effectively defines the release (for example lambdas in Java 8 or modules in Java 9).

With the release of Java 9, however, that release model changed. From Java 10 onward, Oracle decided that Java would be released on a strict, time-based model. This means that OpenJDK now uses a *mainline* development model.

- New features are developed on a branch and merged only when they are code complete
- Releases can occur on a strict time cadence
- Late features do not delay releases but are held over for the next release
- The current head of the trunk should always be releasable (in theory)
- If necessary, an emergency fix can be prepared and pushed out at any point
- Separate OpenJDK projects are used to explore and research longer-term future directions

A new version of Java is released every 6 months ("feature releases"). The various providers (Oracle, Eclipse Adoptium, Amazon, Azul, et al) can choose to make *any* of those releases as a Long-Term Support (LTS) release. However, in practice all of the vendors follow having one release every 3 years being named as the Long-Term Support (LTS) release.

The first LTS release was Java 11, with Java 8 retrospectively included in the set of LTS releases. Oracle's intention was for the Java community to upgrade regularly and to take up the feature releases as they emerge. However, in practice, the community (and enterprise customers in particular) have proved to be resistant to this model - preferring instead to upgrade from one LTS release to the next.

This approach, of course, limits the uptake of new Java features and stifles innovation. However, the realities of enterprise software are what they are, and many people still view an upgrade of Java version as a significant undertaking.

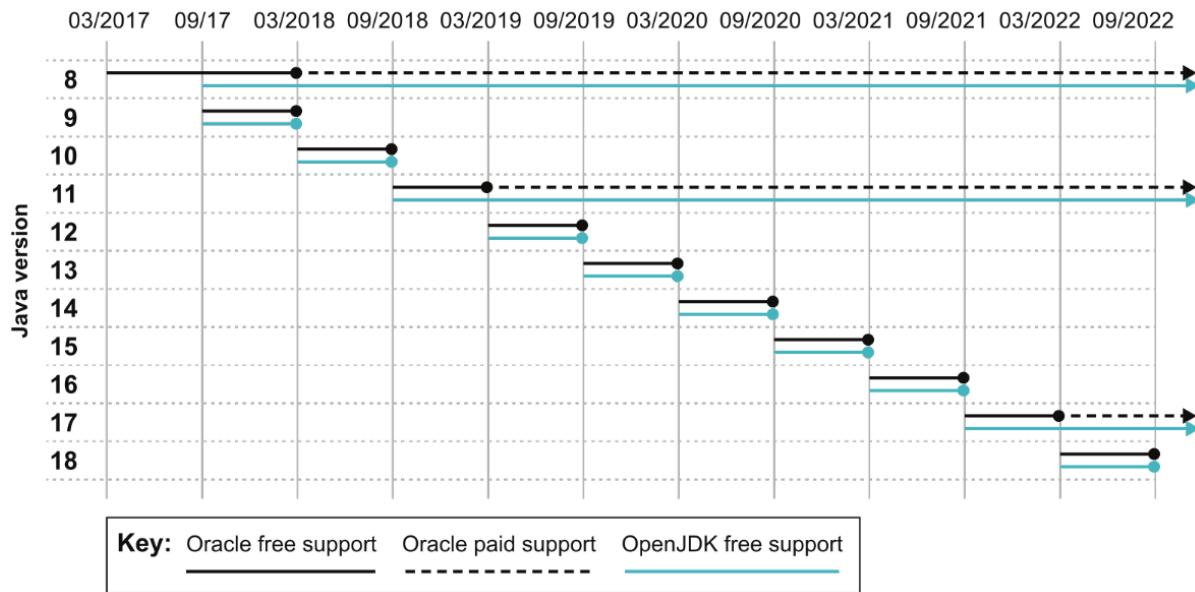


Figure 1.2 The timescale of recent and future releases

This means that whilst the release roadmap is as shown in Figure 1.2, the only releases that have significant usage are the LTS versions - Java 11 (which was released in September 2018) and the previous release, Java 8, which is over 7 years old. These two versions have roughly equal market share, with Java 11 just having overtaken Java 8.

The current LTS release is Java 17, which has just been released (in September 2021). The "classic LTS" release is Java 8 which as of 2021 is still almost half of the Java workloads in the market.

The other significant change in the new release model is that Oracle has changed the license for their distribution. Although Oracle's JDK is built from the OpenJDK sources, the binary is not licensed under an OSS license. Instead, Oracle's JDK is proprietary software, and as of JDK 11, Oracle only provide support and updates for 6 months for each version.

This means that many people who relied on Oracle's free updates are now faced with a choice:

- Pay Oracle for support and updates or
- Use a different distribution that produces open-source binaries

Alternative JDK vendors include Eclipse Adoptium (previously AdoptOpenJDK), Alibaba (Dragonwell), Amazon (Corretto), Azul Systems (Zulu), IBM, Red Hat and SAP.

NOTE

Two of the authors (Martijn & Ben) helped found the AdoptOpenJDK project, which has evolved into the vendor-neutral Eclipse Adoptium community project to build and release a high-quality, free and open-source Java binary distribution - see adoptium.net for more details.

Whilst the Java release model has changed to use timed releases, the vast majority of teams are still running on either JDK 8 or 11. These LTS releases are being maintained by the community (including major vendors), and still receive regular security updates and bug fixes.

The changes made to the LTS versions are deliberately small in scope and are "housekeeping updates". Apart from security and small correctness bug fixes, only a minimal set of changes are permitted. These include fixes needed to make sure that the LTS releases will continue to work correctly for their expected lifetime. This includes things like:

- The addition of the new Japanese Era
- Timezone database updates
- TLS 1.3
- Adding Shenandoah a low-pause GC for large modern workloads

One other necessary change is that the build scripts for macOS needed to be updated to work with a recent version of Apple's Xcode tool so that they will continue to work on new releases of Apple's operating system.

Within the projects to maintain JDK 8 and 11 (sometimes called the "updates" projects), there is still some potential scope for new features to be backported - but it is minimal. As an example, one of the guiding rules is that newly ported features may not change program semantics. Examples of permissible changes could include the support for TLS 1.3 or the backport of Java Flight Recorder to Java 8u272.

Now we've set the scene by clarifying the difference between the language and platform and explaining the new release model, let's meet our first technical feature of modern Java.

The new feature we're going to meet is something that developers have been asking for since almost the first release of Java - a way to reduce the amount of typing that writing Java programs seems to involve.

1.3 Enhanced type inference (var keyword)

Java has historically had a reputation as a verbose language. However, in recent versions, the language has evolved to make more and more use of *type inference*. This is a feature of the source code compiler whereby it doesn't need to be explicitly told all the type information in programs and instead can work some of it out automatically.

NOTE	The aim of type inference is to reduce boilerplate, remove duplication and allow for more concise and readable code.
-------------	---

This trend started with Java 5 when generic methods were introduced. Generic methods permit a very limited form of type inference of generic type arguments, so that instead of having to

explicitly provide the exact type that is needed, like this:

```
List<Integer> empty = Collections.<Integer>emptyList();
```

the generic type parameter can be omitted on the right-hand side:

```
List<Integer> empty = Collections.emptyList();
```

This way of writing a call to a generic method is so familiar that many developers will struggle to remember the form with explicit type arguments. This is a good thing - it means the type inference is doing its job and removing the superfluous boilerplate so that the meaning of the code is clear.

The next significant enhancement to type inference in Java came with version 7, which introduced a change when dealing with generics. Before Java 7, it was common to see code like this:

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

That is a really verbose way to declare that you have some users, whom you identify by `userid` (which is an integer), and each user has a set of properties (modeled as a map of string to strings) specific to that user.

In fact, almost half of the source is duplicated characters, and they don't tell us anything. So, from Java 7 onwards, we can write:

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

and have the compiler work out the type information on the right side. The compiler is working out the correct type for the expression on the right side - it isn't just substituting the text that defines the full type.

NOTE

Because the shortened type declaration looks like a diamond, this form is called "diamond syntax".

In Java 8, more type inference was added to support the introduction of lambda expressions, like this example where the type inference algorithm can conclude that the type of `s` is a `String`:

```
Function<String, Integer> lengthFn = s -> s.length();
```

In modern Java, type inference has been taken one step further, with the arrival of *Local Variable Type Inference* (LVTI), otherwise known as `var`. This feature was added in Java 10 and allows the developer to infer the types of *variables*, instead of the types of *values*, like this:

```
var names = new ArrayList<String>();
```

This is implemented by making `var` a reserved, "magic" type name rather than a language keyword. Developers can still in theory use `var` as the name of a variable, method or package.

NOTE

An important side effect of using `var` appropriately is that the domain of your code is once more front and center (as opposed to the type information). But with great power comes great responsibility! Make sure that you name your variables carefully to help future readers of your code.

On the other hand, code that previously used `var` as the name of a type will have to be recompiled. However, virtually all Java developers follow the convention that type names should start with a capital letter, so the number of instances of pre-existing types called `var` should be vanishing small.

This means that it is entirely legal to write code like that shown in Listing 1.1

Listing 1.1 Bad Code

```
package var;

public class Var {
    private static Var var = null;

    public static Var var() {
        return var;
    }

    public static void var(Var var) {
        Var.var = var;
    }
}
```

and then call it like this:

```
var var = var();
if (var == null) {
    var(new Var());
}
```

However, just because something is *legal*, does not mean it is *sensible*. Writing code like the above is not going to make you any friends and should not pass code reviews!

The intention of `var` is to reduce verbosity in Java code and to be familiar to programmers coming to Java from other languages. It does not introduce dynamic typing, and all Java variables continue to have static types at all times - it's just that it isn't necessary to write them down explicitly in all cases.

Type inference in Java is *local*, and in the case of `var` the algorithm only examines the

declaration of the local variable. This means it cannot be used for fields, method arguments or return types. The compiler applies a form of *constraint solving* to determine whether any type exists that could satisfy all the requirements of the code as written.

For example, in the declaration of `lengthFn` above then the constraint solver can deduce that the type of the method parameter, `s` must be compatible with `String` which is explicitly provided as the type of the parameter to `Function`. In Java, of course, the string type is `final` so the compiler can conclude that the type of `s` is exactly `String`.

For the compiler to be able to infer types, then enough information must be provided by the programmer to allow the constraint equations to be solved. For example, code like this:

```
var fn = s -> s.length();
```

does not have enough type information for the compiler to deduce the type of `fn`, and so it will not compile. One important case of this is:

```
var n = null;
```

which cannot be resolved by the compiler as the null value can be assigned to a variable of any reference type, so there is no information about what types `n` could conceivably be. We say that the type constraint equations that the inferencer needs to solve are "underdetermined" in this case - a mathematical term that connects the number of equations to be solved with the number of variables.

You could imagine a scheme of type inference that goes beyond just the initial declaration of the local variable and examines more code to make inference decisions, like this:

```
var n = null;
String.format(n);
```

A more complex inference algorithm (or a human) might be able to conclude that the type of `n` is actually `String`, as the `format()` method takes a string as the first argument.

This might seem appealing but, as with everything else in software, it represents a trade-off. More complexity means longer compilation times, and a wider variety of ways in which the inference can fail. This, in turn, means that the programmer must develop a more complicated intuition in order to use non-local type inference correctly.

Other languages may choose to make different trade-offs, but Java is clear - only the declaration is used to infer types. Local-variable type inference is intended to be a beneficial technique to reduce boilerplate and verbosity. However, it should be used where necessary to make the code clearer, not as a blunt instrument to be used whenever possible (the "Golden Hammer" anti-pattern).

Some quick guidelines for when to use LVTI:

- In simple initializers - if the right-hand side is a call to a constructor or static factory method
- If removing the explicit type deletes repeated or redundant information
- If variables have names that already indicate their types
- If the scope and usage of the local variable is short and simple

A complete set of applicable rules of thumb is provided by Stuart Marks, one of the core developers of the Java language, in his style guides for LVTI usage:

openjdk.java.net/projects/amber/LVTIstyle.html

To conclude this section, let's look at another, more advanced, usage of `var` - the so-called *non-denotable types*. These are types that are legal in Java, but which cannot appear as the type of a variable. Instead, they must be inferred as the type of the expression that is being assigned.

Let's look at a simple example using the `jshell` interactive environment which arrived in Java 9:

```
jshell> var duck = new Object() {
...>     void quack() {
...>         System.out.println("Quack!");
...>     }
...> }
duck ==> $0@5910e440

jshell> duck.quack();
Quack!
```

The variable `duck` has an unusual type - it is effectively `Object` but extended with a method called `quack()`. While the object may quack like a duck, its type lacks a name, so we can't use the type as either a method parameter or return type.

With LVTI we can use it as the inferred type of a local variable. This allows us to use the type within a method. Of course, the type can't be used outside of this tight local scope, so the overall utility of this language feature is limited - so it's more a curiosity than anything else.

Despite these limitations, this does represent a glimpse at Java's take on a feature that is present in some other languages - sometimes referred to as *structural typing* in statically typed languages and *duck typing* in dynamically typed languages (particularly Python).

1.4 Changing the language and the platform

We think it's essential to explain the "why" of language change as well as the "what." During the development of new versions of Java, there is often much interest around new language features, but the community doesn't always understand how much work is required to get changes fully engineered and ready for prime time.

You may also have noticed that in a mature runtime such as Java, language features tend to evolve from other languages or libraries, make their way into popular frameworks and only then get added to the language or runtime itself.

We hope to shed a bit of light on this area, and hopefully dispel a few myths along the way. But if you're not very interested in how Java evolves, feel free to skip ahead to section 1.5 and jump right into the language changes.

There is an effort curve involved in changing the Java language—some possible implementations require less engineering effort than others. In figure 1.3, we've tried to represent the different routes and show the relative effort required for each.

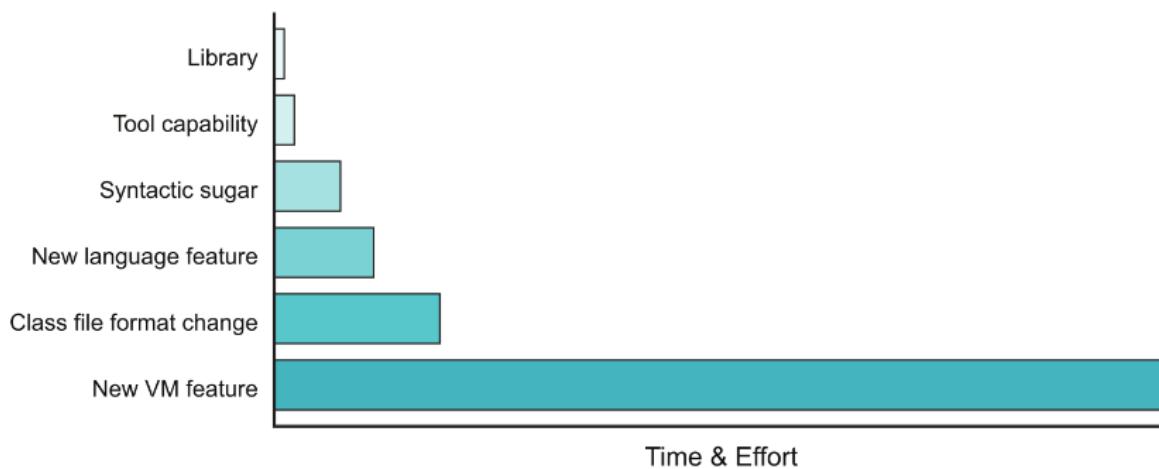


Figure 1.3 The relative effort involved in implementing new functionality in different ways

In general, it's better to take the route that requires the least effort. This means that if it's possible to implement a new feature as a library, you generally should. But not all features are easy, or even possible, to implement in a library or an IDE capability. Some features have to be implemented deeper inside the platform.

Here's how some recent features fit into our complexity scale for new language features:

- *Library change* — Collections factory methods (Java 9)
- *Syntactic sugar* — Underscores in numbers (Java 7)
- *Small new language feature* — try-with-resources (Java 7)
- *Class file format change* — Annotations (Java 5)
- *New JVM feature* — Nestmates (Java 11)
- *Major new feature* — Lambda Expressions (Java 8)

Let's take a close look at how changes across the complexity scale get made.

1.4.1 Sprinkling some sugar

A phrase that's sometimes used to describe a language feature is "syntactic sugar". That is, the syntactic sugar form is provided because it's easier for humans to work with despite the functionality already existing in the language.

As a rule of thumb, a feature referred to as syntactic sugar is removed from the compiler's representation of the program early in the compilation process—it's said to have been "desugared" into the basic representation of the same feature.

This makes syntactic sugar changes to a language easier to implement because they usually involve a relatively small amount of work, and only involve changes to the compiler (`javac` in the case of Java).

One question that might well be asked at this point is, "What constitutes a small change to the spec?" One of the most straightforward changes in Java 7 consisted of adding a single word—"String"—to section 14.11 of the JLS, which allowed strings in a switch statement. You can't really get much smaller than that as a change, and yet even this change touches several other aspects of the spec. Any alteration produces consequences, and these have to be chased through the entire design of the language.

1.4.2 Changing the language

The full set of actions that must be performed (or at least investigated) for *any* change is as follows:

- Update the JLS
- Implement a prototype in the source compiler
- Add library support essential for the change
- Write tests and examples
- Update documentation

In addition, if the change touches the VM or platform aspects:

- Update the VMSpec
- Implement the VM changes
- Add support in the class file and VM tools
- Consider the impact on reflection
- Consider the impact on serialization
- Think about any effects on native code components, such as Java Native Interface (JNI).

This isn't a small amount of work, and that's after the impact of the change across the whole language spec has been considered!

An area of hairiness, when it comes to making changes, is the type system. That isn't because

Java's type system is terrible. Instead, languages with rich static type systems are likely to have a lot of possible interaction points between different bits of those type systems. Making changes to them is prone to creating unexpected surprises.

1.4.3 JSRs and JEPs

There are two main mechanisms that are used to make changes to the Java platform. The first is the *Java Specification Request* (JSR), which is specified by the *Java Community Process* (JCP). This is used to determine standard APIs - both external libraries and major internal platform APIs.

This was historically the only way of making changes to the Java platform, and was best used to codify a consensus of already mature technology. However, in recent years, a desire to implement change faster (and in smaller units) led to the development of the JDK Enhancement Proposal (JEP) as a lighter-weight alternative. Platform (or umbrella) JSRs are now made up of JEPs that are targeted for that version, with the JSR process gives extra intellectual property protections for the whole ecosystem.

When discussing new Java features, it is often useful to refer to an upcoming or recent feature by its JEP number - and a complete list of all JEPs - including those that have been delivered or withdrawn can be found at openjdk.java.net/jeps/0.

1.4.4 Incubating and preview features

Within the new release model, Java now has two mechanisms for trying out a proposed feature before finalizing it in a later release. The aim of these mechanisms is to provide better features by gathering feedback from a much wider pool of users, and potentially to change or withdraw the feature before it becomes a permanent part of Java.

Incubating Features are new APIs and their implementation, which in their simplest form are effectively just a new API shipped as a self-contained module (we will meet the details of Java modules in Chapter 2). The name of the module is chosen so that it makes it clear that the API is temporary and will change when the feature is finalized.

NOTE

This means that any code that relies upon a non-finalized version of an incubating feature will have to make changes when the feature becomes final.

One very visible example of an incubating feature is the new support for version 2 of the HTTP protocol, usually referred to as HTTP/2. In Java 9, this was shipped as the incubator module `jdk.incubator.http`. The naming of this module, and the use of the `jdk.incubator` namespace rather than `java` clearly marked the feature as non-standard and subject to change.

The feature was standardized in Java 11 when it was moved to the `java.net.http` module in the `java` part of the namespace.

NOTE

We will meet another incubating feature in Chapter 17 when we discuss the Foreign Access API, which is part of an OpenJDK project codenamed Panama.

The main advantage of this approach is that an incubating feature can be isolated to a single namespace. Developers can quickly try out the feature, and even use it in production code - providing they are happy to modify some code, recompile and relink when the feature becomes standardized.

Preview Features are the other mechanism that recent Java versions provide for shipping non-finalized features. They are more intrusive than incubating features as they are implemented as part of the language itself, at a deeper level. These features potentially require support from:

- The `javac` compiler
- Bytecode format
- Class file & classloading

They are only available if specific flags are passed to the compiler and runtime. Trying to use preview features without the flags enabled is an error - both at compile time and at runtime.

This makes them much more complex to handle (compared to incubating features). As a result, preview features can't really be used in production. For one thing they are represented by a version of the classfile format that is not finalized and may never be supported by any production version of Java.

This means that preview features are only suitable for experimentation, developer testing and familiarization. Unfortunately, in almost all deployments, only fully finalized features can be used in code that is destined for production.

Java 11 did not contain any preview features (although a first preview version of *switch expressions* arrived in Java 12), so it's hard to give a good example of one in this section. We'll dig more into preview versions in Chapter 3 when we discuss Java 17, though.

1.5 Small changes in Java 11

Since Java 8, a relatively large number of new small features have appeared in successive releases. Let's take a quick tour through some of the most important ones - although this is by no means all the changes.

1.5.1 Collections factories (JEP 213)

An often-requested enhancement is to extend Java to support a simple way to declare *collection literals* - a dumb collection of objects (such as a list or a map).

This seems attractive as many other languages support some form of this, and Java itself has always had array literals:

```
jshell> int[] numbers = {1, 2, 3};
numbers ==> int[3] { 1, 2, 3 }
```

However, although it seems superficially attractive, adding this feature at language level has some significant drawbacks. For example, although `ArrayList`, `HashMap` and `HashSet` are the implementations that are most familiar to developers, a primary design principle of the Java Collections are that they are represented as interfaces, not classes. Other implementations are available and are widely used.

This means that it would run counter to the design intent to have a new syntax that directly couples to specific implementations, no matter how common.

Instead, the design decision was to add simple factory methods to the relevant interfaces, exploiting the fact that Java 9 added the ability to have static methods on interfaces. The resulting code looks like this:

```
Set<String> set = Set.of("a", "b", "c");
var list = List.of("x", "y");
```

and whilst this is a little more verbose than adding support at language level, the complexity cost in implementation terms is substantially less.

These new methods are implemented as a set of overloads:

```
List<E> List<E>.of()
List<E> List<E>.of(E e1)
List<E> List<E>.of(E e1, E e2)
List<E> List<E>.of(E e1, E e2, E e3)
List<E> List<E>.of(E e1, E e2, E e3, E e4)
List<E> List<E>.of(E e1, E e2, E e3, E e4, E e5)
List<E> List<E>.of(E e1, E e2, E e3, E e4, E e5, E e6)
List<E> List<E>.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
List<E> List<E>.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
List<E> List<E>.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
List<E> List<E>.of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9,
    E e10)
List<E> List<E>.of(E... elements)
```

The common cases (up to 10 elements) are provided, along with a varargs form for the unlikely use case that more than ten elements are required in the collection.

For maps, the situation is a little more complicated, because maps have two generic parameters

(the key type and the value type) and so, whilst the simple cases can be written like this:

```
var m1 = Map.of(k1, v1);
var m2 = Map.of(k1, v1, k2, v2);
```

There is no simple way of writing the equivalent of the varargs form for map, so instead a different factory method `ofEntries()` is used in combination with a static helper method `entry()` to provide an equivalent of a varargs form:

```
Map.ofEntries(
    entry(k1, v1),
    entry(k2, v2),
    // ...
    entry(kn, vn));
```

One final point that developers should be aware of - the factory methods produce instances of immutable types:

```
jshell> var ints = List.of(2, 3, 5, 7);
ints ==> [2, 3, 5, 7]

jshell> ints.getClass();
$2 ==> class java.util.ImmutableCollections$ListN
```

These class are new implementations of the Java Collections interfaces that are immutable - they are not the familiar, mutable classes (such as `ArrayList` and `HashMap`). Attempts to modify instances of these types will result in an exception being thrown.

1.5.2 Remove enterprise modules (JEP 320)

Over time, Java Standard Edition (aka Java SE) had a few modules added to it that were really part of Java Enterprise Edition (Java EE) such as:

- JAXB
- JAX-WS
- CORBA
- JTA

In Java 9, the packages that implemented these technologies were moved into non-core modules and deprecated for removal.

- `java.activation` (JAF)
- `java.corba` (CORBA)
- `java.transaction` (JTA)
- `java.xml.bind` (JAXB)
- `java.xml.ws` (JAX-WS, plus some related technologies)
- `java.xml.ws.annotation` (Common Annotations)

As part of an effort to slimline the platform, in Java 11 these modules have been removed. Three

related modules used for tooling and aggregation have also been removed from the core SE distribution.

- `java.se.ee` (Aggregator module for the six modules above)
- `jdk.xml.ws` (Tools for JAX-WS)
- `jdk.xml.bind` (Tools for JAXB)

Projects built on Java 11 and later that want to use these capabilities now require the inclusion of an explicit external dependency. This means that some programs that relied upon these APIs built cleanly under Java 8 but require modifications to their build script in order to build under Java 11.

We will investigate this specific issue more fully in Chapter 11.

1.5.3 HTTP/2 (Java 11)

In modern times, a new version of the HTTP standard has been released - HTTP/2. We're going to examine the reasons for finally updating the venerable HTTP 1.1 specification (dating from 1997!). Then we'll see how Java 11 gives the well-grounded developer access to the new features and performance of HTTP/2.

As you might expect for technology from 1997, HTTP 1.1 has been showing its age, particularly around performance in modern web applications. Limitations include problems such as:

- Head-of-line blocking
- Restricted connections to a single site
- Performance overhead of HTTP control headers

HTTP/2 is a *transport* level update to the protocol focused on fixing these sorts of fundamental performance issues that don't fit how the web really works today.

With its performance focus on how bytes flow between client and server, HTTP/2 actually doesn't alter many of the familiar HTTP concepts—request/response, headers, status codes, response bodies—all remain semantically the same in HTTP/2 vs HTTP 1.1.

HEAD-OF-LINE BLOCKING

Communication in HTTP takes place over TCP sockets. While HTTP 1.1 defaulted to reusing individual sockets to avoid repeating unnecessary setup costs, the protocol dictated that requests returned in order even when multiple requests shared a socket (known as pipelining). This means that a slow response from the server blocked subsequent requests which theoretically could have been returned sooner.

These effects are readily visible in places like browser rendering stalling on downloading assets. The same one-response-per-connection-at-a-time behavior can also limit JVM applications talking to HTTP-based services.

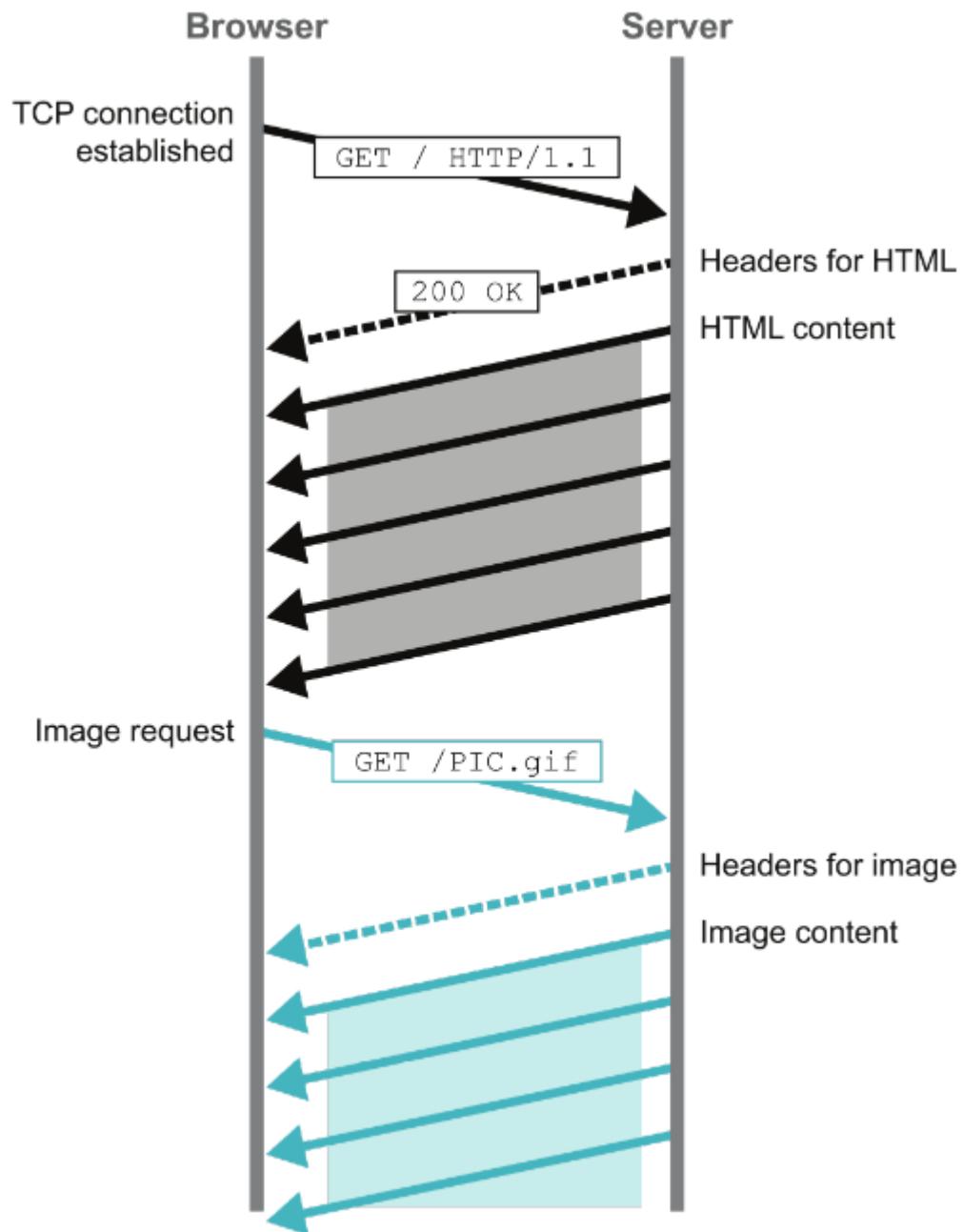


Figure 1.4 HTTP/1.1 transfers

HTTP/2 is designed from the ground up to multiplex requests over the same connection. Multiple *streams* between the client and server are always supported. It even allows for separately receiving the headers and the body of a single request.

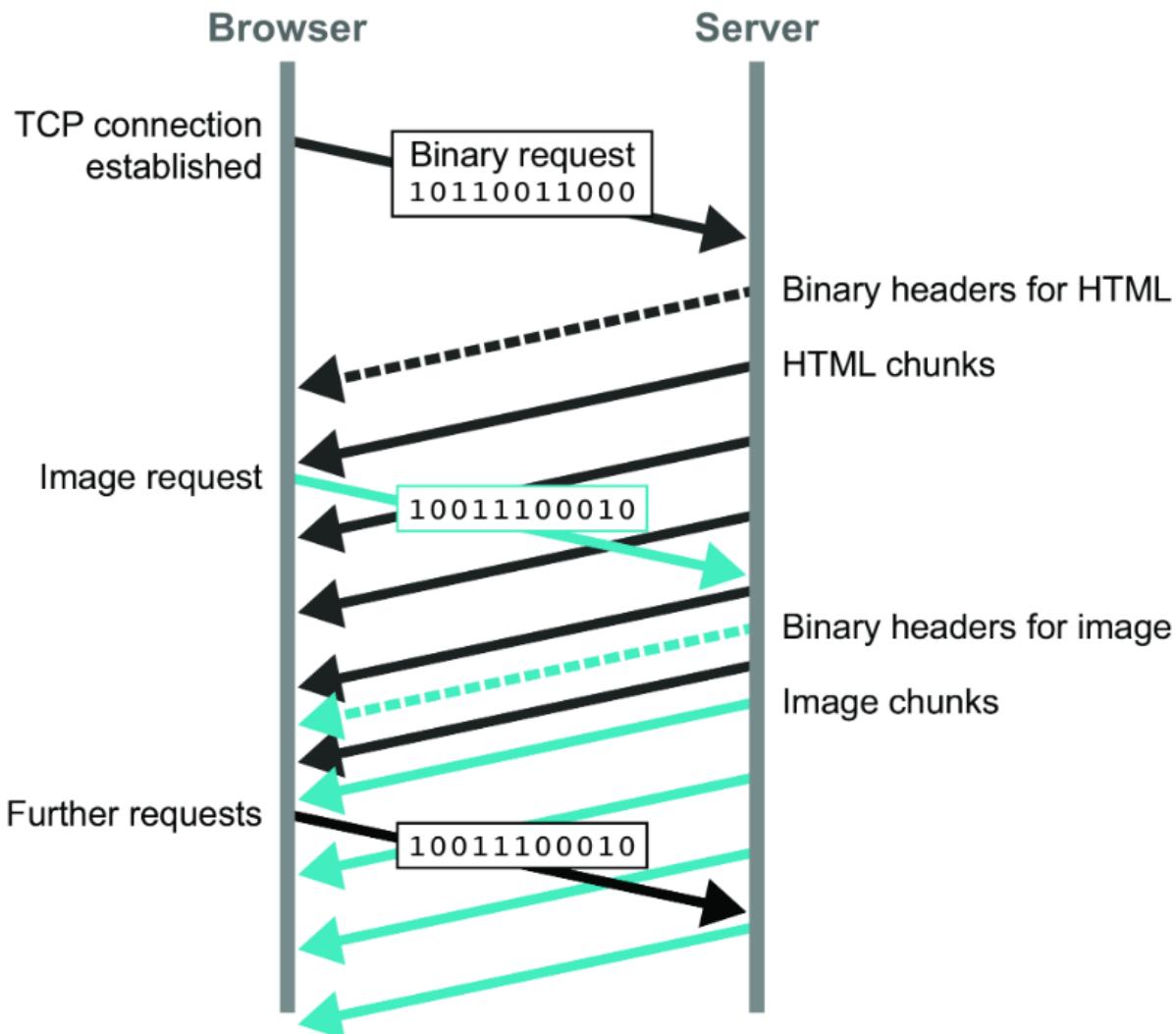


Figure 1.5 HTTP/2 transfers

This fundamentally changes assumptions that decades of HTTP 1.1 have made second nature to many developers. For instance, it's long been accepted that returning lots of small assets on a website performed worse than making larger bundles. JavaScript, CSS and images all have common techniques and tooling for smashing many smaller files together to return more efficiently. In HTTP/2, multiplexed responses mean your resources don't block behind other slow requests, and smaller responses may be more accurately cached yielding a better experience overall.

RESTRICTED CONNECTIONS

The HTTP 1.1 specification recommends limiting connections to a server to only 2 at a time. This is listed as a **SHOULD** rather than a **MUST**, and modern web browsers often allow between 6-8 connections per domain. This limit to concurrent downloads from a site has often led developers to serve sites from multiple domains or implement the sort of bundling mentioned before.

HTTP/2 addresses this because each connection can effectively be used to make as many simultaneous requests as desired. Browsers only open one connection to given domain, but can perform many requests over that same connection at the same time.

In our JVM applications where we might have pooled HTTP 1.1 connections to allow for more concurrent activity, HTTP/2 gives us another built-in way to squeeze more requests out.

HTTP HEADER PERFORMANCE

A significant feature of HTTP is the ability to send *headers* alongside requests. Headers are a critical part of how the HTTP protocol itself is stateless, but our applications can maintain state between requests (such as the fact your user is logged in).

While the body of HTTP 1.1 payloads may be compressed if the client and server can agree on the algorithm (typically gzip), headers don't participate. As richer web applications make more and more requests, the repetition of increasingly large headers can be a problem, especially for larger websites.

HTTP/2 addresses this with a new binary format for headers. As a user of the protocol you don't have to think much about this - it's simply built into how headers are transmitted between client and server.

TLS ALL THE THINGS

HTTP 1.1 in 1997 entered a very different internet than we see today. Commerce on the internet was only starting to take off, and security wasn't always a top concern in early protocol designs. Computing systems were also slow enough to make practices like encryption often far too expensive.

HTTP/2 was officially accepted in 2015 into a world far more security conscious. In addition, the computing needs for ubiquitous encryption of web requests through TLS (known in earlier versions as SSL) are low enough to have removed most argument over whether to encrypt or not. As such, in practice, HTTP/2 is only supported with TLS encryption (the protocol does, in theory, allow for transmission in cleartext but none of the major implementations provides it).

This has an operational impact on deploying HTTP/2, as it requires a certificate with a lifecycle of expiration and renewal. For enterprises, this increases the need for certificate management. [Let's Encrypt](#), and other private options have been growing in response to this need.

OTHER CONSIDERATIONS

While the future is trending toward the uptake of HTTP/2, deployment of it across the web hasn't been fast. In addition to the encryption requirement, which even impacts local development, this may be attributable to some rough edges and extra complexity:

- HTTP/2 is binary-only, working with an opaque format is challenging.
- HTTP layer products such as load-balancers, firewalls, debugging tools require updates to support HTTP/2.
- Performance benefits are aimed mainly at the browser-based use of HTTP. Backend services working over HTTP may see less benefit to updating.

HTTP/2 IN JAVA 11

The arrival of a new HTTP version after so many years motivated JEP 110 to introduce an entirely new API. Within the JDK this replaces (but doesn't remove) `HttpURLConnection` while aiming to put a usable HTTP API "in the box" as it were since many developers have reached for external libraries to fulfill their HTTP related needs.

The resulting HTTP/2 and web socket compatible API came first to Java 9 as an Incubating Feature. JEP 321 moved it to its permanent home in Java 11 under `java.net.http`. The new API supports HTTP 1.1 as well as HTTP/2 and can fall back to HTTP 1.1 when a server being called doesn't support HTTP/2.

Interactions with the new API start from the `HttpRequest` and `HttpClient` types. These are instantiated via builders, setting configurations before issuing the actual HTTP call.

```
var client = HttpClient.newBuilder().build();①  
var uri = new URI("https://google.com");  
var request = HttpRequest.newBuilder(uri).build();②  
var response = client.send(③  
    request,  
    HttpResponse.BodyHandlers.ofString(Charset.defaultCharset()));④  
System.out.println(response.body());
```

- ① Construct an `HttpClient` instance we can use to make requests
- ② Construct a specific request to Google with an `HttpRequest` instance
- ③ Synchronously make the HTTP request and save its response. This line blocks until the entire request has completed.
- ④ The `send` method needs a handler to tell it what to do with the response body. Here we use a standard handler to return the body as a `String`.

This demonstrates the synchronous use of the API. After building our request and client, we issue the HTTP call with the `send` method. We won't receive the `response` object back until the full HTTP call has completed, much like the older HTTP APIs in the JDK.

The first parameter is the request we set up, but the second deserves a closer look. Rather than expect to always return a single type, the `send` method expects us to provide an implementation of `HttpResponse.BodyHandler<T>` interface to tell it how to handle the response.

`HttpResponse.BodyHandlers` provides some useful basic handlers for receiving your response as a byte array, as a string, or as a file. But customizing this behavior is just an implementation of `BodyHandler` away. All of this plumbing is based on `java.util.concurrent.Flow` publisher and subscriber mechanisms, a form of programming known as reactive streams. We'll examine this pattern in more detail in the concurrency chapter of this book.

One of the most significant benefits of HTTP/2 is its built-in multiplexing. Only using a synchronous `send` doesn't really gain those benefits, so it should come as no surprise that `HttpClient` also supports a `sendAsync` method. `sendAsync` returns a `CompletableFuture` wrapped around the `HttpResponse`, providing a rich set of capabilities that may be familiar from other parts of the platform.

```
var client = HttpClient.newBuilder().build();

var uri = new URI("https://google.com");
var request = HttpRequest.newBuilder(uri).build();           ①

var handler = HttpResponse.BodyHandlers.ofString();
CompletableFuture.allOf(
    client.sendAsync(req, handler)                         ②
        .thenAccept((resp) -> System.out.println(resp.body())),
    client.sendAsync(req, handler)                         ③
        .thenAccept((resp) -> System.out.println(resp.body())),
    client.sendAsync(req, handler)                         ④
        .thenAccept((resp) -> System.out.println(resp.body()))
).join();                                                 ⑤
```

- ① Create client and request as before.
- ② Use `CompletableFuture.allOf` to wait for all the requests to finish
- ③ `sendAsync` starts an HTTP request but returns a future and does not block.
- ④ When the future completes, we use `thenAccept` to receive the response.
- ⑤ We can reuse the same client to make multiple requests simultaneously.

Here we set up a request and client again, but then we asynchronously repeat the call three separate times. `CompletableFuture.allOf` combines these three futures so we can wait on them all to finish with a single `join`.

This only scratches the two main entry points to this API. It offers tons of features and customization, from the configuration of timeouts and TLS, all the way to advanced asynchronous features like receiving HTTP/2 server pushes via `HttpResponse.PushPromiseHandler`.

Building off the futures and reactive-streams, the new HTTP API in the JDK provides an attractive alternative to the large libraries which have dominated the ecosystem in the HTTP space. Designed with modern asynchronous programming at the forefront, `java.net.http` puts Java in an excellent place for wherever the web evolves to in the future.

1.5.4 Single-file source-code programs (JEP 330)

The usual way that Java programs are executed is by compiling source code to a class file and then starting up a virtual machine process that acts as an execution container to interpret the bytecode of the class.

This is very different from languages like Python, Ruby and Perl, where the source code of a program is interpreted directly. The Unix environment has a long history of these types of *scripting languages* but Java has not traditionally been counted among them.

With the arrival of JEP 330, Java offers a new way to execute programs. Source code can be compiled in-memory and then executed by the interpreter without ever producing a .class file on disk.

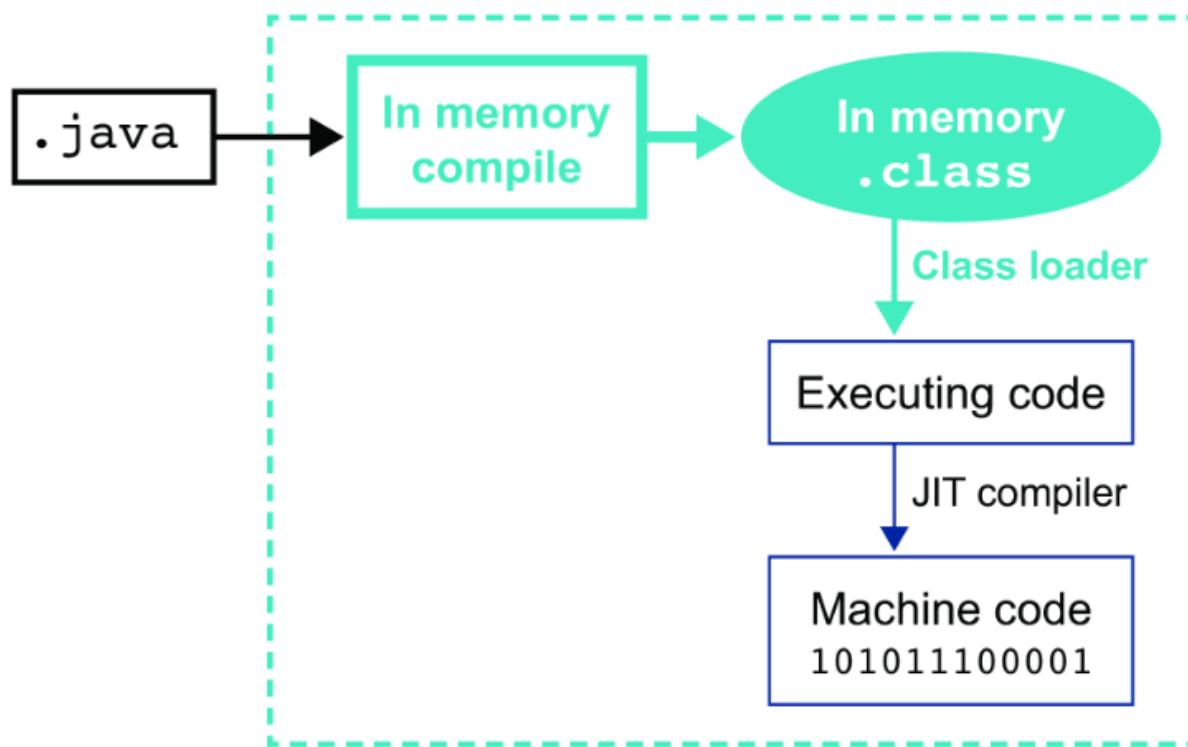


Figure 1.6 Single file execution

This gives a user experience that is like Python and other scripting languages.

The feature has some limitations, specifically that it:

- Is limited to code that lives in a single source file
- Cannot compile additional source files in the same run
- May contain any number of classes in the source file
- Must have the first class declared in the source file as the entry point
- Must define the main method in the entry point class

The feature also uses a `--source` flag to indicate source code compatibility mode - essentially the language level of the script.

Java file naming conventions must be followed for execution - so the class name should match the file name. However, the `.java` extension should **not** be used as this can confuse the launcher.

These types of Java scripts can also contain a shebang line.

```
#!/usr/bin/java --source 11

public final class HTTP2Check {
    public static void main(String[] args) {
        if (args.length < 1) {
            usage();
        }
        // implementation of our HTTP callers... ①
    }
}
```

- ① Full code for `HTTP2Check` provided in project resources

The shebang line provides the necessary parameters so that the file can be marked executable and directly invoked, like this:

```
$ ./HTTP2Check https://www.google.com
https://www.google.com: HTTP_2
```

Whilst this feature does not bring the full experience of scripting languages to Java; it can be a useful way of writing simple, useful tools in the Unix tradition without introducing another programming language into the mix.

1.6 Summary

- The Java language and platform are two separate (if strongly related) components of the Java ecosystem. The platform supports many languages beyond just Java.
- After Java 8, the Java platform has adopted a new timed release process. New versions arrive every six months and a long-term support (LTS) release every 2 or 3 years
- The current LTS versions are 11 and 17, with Java 8 still being supported for now
- With its focus on backward compatibility, making changes to Java can often be difficult. Changes restricted to just the library or compiler are often much simpler than changes that also require updates in the virtual machine.
- Java 11 introduced many useful features that are worth upgrading for:
 - `var` keyword to streamline variable definitions
 - Factory methods to simplify creating lists, maps, and other collections
 - A new `HttpClient` implementation with full HTTP/2 support
 - Single-file programs can be run directly without compiling to class files
 - Switch expressions bring new safety and expressiveness to the `switch` keyword

Java modules

This chapter covers

- Java's platform modules
- Changes to access control semantics
- Writing modular applications
- Multi-release Jars

As mentioned in Chapter 1, versions of Java, up to and including Java 9, were delivered according to a feature-driven release plan, often with a major new capability that defined or was strongly associated with the release.

For Java 9, this feature was Java Platform Modules (also known as JPMS, Jigsaw or just "modules"). This is a major enhancement and change to the Java platform that had been discussed for many years - it was originally conceived of as potentially shipping as a part of Java 7, back in 2009/10.

In this chapter, we will explain the reasons why modules are needed, as well as the new syntax used to articulate modularity concepts and how to use them in your applications. This will enable you to use JDK and third-party modules in your build as well as packaging apps or libraries as modules.

This change has profound implications for the architecture of applications, and modules have many benefits to modern projects that are concerned about things like: process footprint, startup cost and warmup time.

Modules can also help to solve the so-called "JAR Hell" problem that can plague Java applications with complex dependencies.

2.1 Setting the scene

A *module* is a fundamentally new concept in the Java language (as of Java 9). It is a unit of application deployment and dependency which has semantic meaning to the runtime. This is different to existing concepts in Java because:

- Jar files are invisible to the runtime - they're basically just zipped directories containing classfiles.
- Packages are really just namespaces to group classes together for access control
- Dependencies are defined at the class level only
- Access control and reflection combine in a way that produces a fundamentally open system without clear deployment unit boundaries and with minimal enforcement

Modules, on the other hand:

- Define dependency information between modules, so all sorts of resolution and linkage problems can be detected at compile or application start time
- Provide proper encapsulation - so internal packages and classes can be made safe from pesky users who might want to fiddle with them
- Are a proper unit of deployment with metadata that can be understood and consumed by a modern Java runtime and are reflected in the Java type system (e.g. reflectively)

NOTE

Before modules, within the core language and runtime environment there is no aggregated dependency metadata. Instead, it is defined only in build systems like Maven or in third-party modules systems (such as OSGI or JBoss Modules) that the JVM neither knows nor cares about.

Java platform modules represent an implementation of a missing concept within the Java world as it existed at version 8.

NOTE

Java modules are often packaged as special jar files - but they are not tied to that format (and there are other possible formats as we will see).

The aim of the modules system is to make the deployment units (modules) as independent of each other as possible. The idea is that modules are able to be separately loaded and linked, although in practice, real applications may well end up depending on a group of modules that provide related capabilities (such as security).

2.1.1 Project Jigsaw

The project within OpenJDK to deliver the modules feature was known as *Project Jigsaw*. It aimed to deliver a full-featured modularity solution which included these goals:

- Modularisation of the JDK platform source
- Reduce process footprint size
- Improve application startup time
- Make modules available to both the JDK and to application code
- Allow true strict encapsulation for the first time in Java
- Add new, previously impossible, access control modes to the Java language

These goals were, in turn, driven by other objectives that are more closely focused on the JDK and Java runtime:

- The end of a single, monolithic `rt.jar`
- Properly encapsulate and protect JDK internals
- Allow major (& breaking) internal changes to be made
- Introduce modules as "super packages"

These secondary goals may require a bit more explanation as they are more closely connected to internal and implementation aspects of the platform.

MODULAR, NOT MONOLITHIC JAVA RUNTIME

The legacy JAR format is essentially just a zip file that contains classes. It dates back to the earliest days of the platform and is in no way optimized for Java classes and applications. Abandoning the JAR format for the platform classes can help in several areas - for example enabling much better startup performance.

Modules provide two new formats - *JMOD* and *JIMAGE* - which are used at different times (compile / link time and runtime, respectively) in the program lifecycle.

The jmod format is somewhat similar to the existing JAR format, but it has been modified to allow the inclusion of native code as part of a single file (rather than having to ship a separate shared object file as is done in Java 8).

NOTE	For most developers needs, including publishing modules to Maven, it's better to package your own modules as modular jars rather than as jmods.
-------------	--

The jimage format is used to represent a Java runtime image. Until Java 8 there were only two possible runtime images (JDK and JRE) but this was largely an accident of history. Oracle introduced the *Server JRE* with Java 8 (as well as *compact profiles*) as a stepping stone towards full modularity.

In a modular application there is enough metadata that the exact set of dependencies can be known before program start. This leads to the possibility that only what is needed has to be loaded - which is much more efficient.

It is possible to go even further and define a *custom runtime image* that can be shipped along

with an application and which does not contain a full, general purpose installation of Java but only what the application requires. We will encounter this last possibility at the end of this chapter when we meet the `jlink` tool.

For now, let's meet the `jimage` tool that's available to show details about a Java runtime image. For example, for a Java 15 full runtime (i.e. what used to be contained in a JDK):

```
$ jimage info $JAVA_HOME/lib/modules
Major Version: 1
Minor Version: 0
Flags: 0
Resource Count: 32780
Table Length: 32780
Offsets Size: 131120
Redirects Size: 131120
Locations Size: 680101
Strings Size: 746471
Index Size: 1688840
```

or

```
$ jimage list $JAVA_HOME/lib/modules
jimage: /Library/Java/JavaVirtualMachines/java15/Contents/Home/lib/modules

Module: java.base
    META-INF/services/java.nio.file.spi.FileSystemProvider
    apple/security/AppleProvider$1.class
    apple/security/AppleProvider$ProviderService.class
    apple/security/AppleProvider.class
    apple/security/KeychainStore$CertKeychainItemPair.class
    apple/security/KeychainStore$KeyEntry.class
    apple/security/KeychainStore$TrustedCertEntry.class
    apple/security/KeychainStore.class
    com/sun/crypto/provider/AESCipher$AES128_CBC_NoPadding.class
    ... many, many lines of output
```

Moving away from `rt.jar` allows for better startup performance and to optimize for only what is needed by an application. The new formats are designed to be opaque to the developer, and are implementation-dependent. It is no longer possible to just unzip `rt.jar` and get back the JDK's class library.

This is just one step, however, in making the platform's internals less accessible to Java programmers, which was one of the goals of the modules system.

ENCAPSULATE THE INTERNALS

The contract between the Java platform and its users was always intended to be an API contract - that backwards compatibility would be maintained at the interface level, **not** in the details of the implementation.

However, Java developers have not held up their end of the bargain and instead, over time, have tended to use parts of the platform implementation that were never intended for public consumption.

This is problematic, because the OpenJDK platform developers want the freedom to modify the implementation of the JVM and platform classes to future-proof and modernize them - to provide new features and better performance without worrying about breaking user applications.

One major impediment to making breaking changes to the platform internals is Java's approach to access control as it exists in Java 8. Java only defines `public`, `private`, `protected` and `package-private` as access control levels, and these modifiers are only applied at class level and finer.

There are numerous ways to work around these restrictions (such as reflection, or creating additional classes in relevant packages) and there is no foolproof (or expert-proof) way to fully protect the internals.

The use of the workarounds to access the internals was historically often for valid reasons. As the platform has matured however, an official way of accessing almost all of the desired functionality has been added. The unprotected internals therefore represent a liability for the platform going forward without a corresponding benefit - and modularity was one way to remove that legacy problem.

To sum up, Project Jigsaw was a way to solve several problems at once - primarily to reduce runtime size, improve startup time and tidy up dependencies between internal packages. These were problems that were hard (or impossible) to tackle incrementally. Opportunities for these types of "non-local" improvement do not come along very often, especially in mature software platforms, so the Jigsaw team wanted to take advantage of their circumstances.

THE JVM IS MODULAR NOW

To see this, consider this very simple program:

```
public class StackTraceDemo {
    public static void main(String[] args) {
        var i = Integer.parseInt("Fail");
    }
}
```

Compiling and running this code produces a runtime exception:

```
$ java StackTraceDemo
Exception in thread "main" java.lang.NumberFormatException:
For input string: "Fail"
  at java.base/java.lang.NumberFormatException.forInputString(
  NumberFormatException.java:65)
  at java.base/java.lang.Integer.parseInt(Integer.java:652)
  at java.base/java.lang.Integer.parseInt(Integer.java:770)
  at StackTraceDemo.main(StackTraceDemo.java:3)
```

However, we can clearly see that the format of the stack trace has changed somewhat from the form that was used in Java 8. In particular, the stack frames are now qualified by a module name

(`java.base`) as well as a package name, class name and line number. This clearly shows that the modular nature of the platform is pervasive and is present for even the simplest program.

2.1.2 The module graph

Key to all of modularity is the *module graph*, which is a representation of how modules depend on each other. Modules make their dependencies explicit via some new syntax and those dependencies are hard guarantees that the compiler and runtime can rely upon. One very important concept is that the module graph must be a *directed acyclic graph* (DAG) so in mathematical terms there cannot be any cyclic dependencies.

NOTE It is important to realise that in modern Java environments, all applications run on top of the modular JRE; there is not a "modular mode" and a "legacy classpath mode".

Although not every developer needs to become an expert in the modules system, it makes sense that a well-grounded Java developer would benefit from a working knowledge of a new subsystem that has changed the way that all programs are executed on the JVM.

Let's take a look at a first view of the modules system as most developers encounter it.

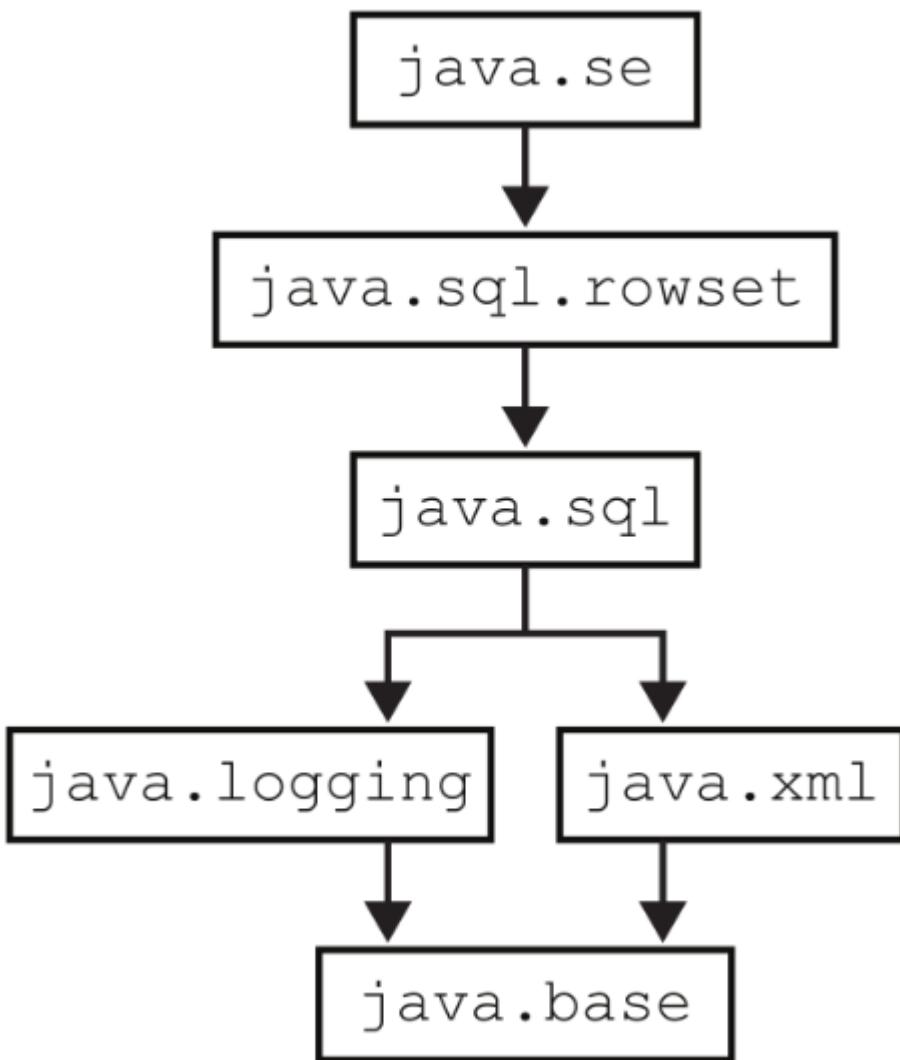


Figure 2.1 JDK System Modules (simplified view)

In Figure 2.1 we can see a simplified view of some of the main modules in the JDK. Note that the module `java.base` is always a dependency of every module. When drawing pictures of module graphs the implicit dependency on `java.base` is often eliminated just to reduce visual clutter.

The clean and relatively simple set of module boundaries that we can see in Figure 2.1 needs to be contrasted with the state of the JDK in Java 8. Unfortunately, before modules, Java's top-level unit of code was the package - and Java 8 had almost 1000 of them in the standard runtime. This would be essentially impossible to draw, and the dependencies within the graph would be so complex that a human would not be able to make sense of it.

Taking the pre-modular JDK and reshaping it into the well-defined form that we see today was not easy to achieve, and the path to delivering JDK modularity was long.

Java 9 was released in September 2017 but development of the feature had begun several years

before that with the Java 8 release train. In particular, there were several sub-goals that were necessary first steps for the delivery of modules, including:

- Modularize the layout of source code in the JDK (JEP 201)
- Modularize the structure of run-time images (JEP 220)
- Disentangle complex implementation dependencies between JDK packages

Even though the finished modules feature did not ship until Java 9, much of the cleanup was undertaken as part of Java 8, and even allowed a feature known as *compact profiles* (which we will meet at the end of this chapter) to ship as part of that release.

2.1.3 Protecting the internals

One of the major problems that modules needed to solve was over-coupling of user Java frameworks to internal implementation details. For example, this piece of Java 8 code extends an internal class to get access to a low-level *URL canonicalizer*.

A URL canonicalizer is a piece of code that takes a URL in one of the various forms permitted by the URL standard and converts it to a standard (canonical) form. The intent is that canonical URLs can act as a single source of truth for the location of content that can be accessed via multiple different possible URLs.

The following code is for demonstration purposes only, so we can have a concrete example to discuss modules and access control - your code should never access internal classes directly.

```
import sun.net.URLCanonicalizer;

public class MyURLHandler extends URLCanonicalizer {

    public boolean isSimple(String url) {
        return isSimpleHostName(url);
    }
}
```

If we try to compile it using Java 8, `javac` warns us that we're accessing an internal API:

```
$ javac AccessInternals.java
AccessInternals.java:1: warning: Version is internal proprietary API and
    may be removed in a future release
    import sun.misc.Version;
                           ^
AccessInternals.java:5: warning: Version is internal proprietary API and
    may be removed in a future release
        Version.println();
                           ^
2 warnings
```

However, by default the compiler still allows the access, and the result is a user class that is tightly coupled to the internal implementation of the JDK. This is fragile and will break if the called code moves or is replaced.

If enough developers abuse this openness, then this leads to a situation in which it is difficult or impossible to make changes to the internals, as to do so would break deployed libraries and applications.

NOTE

The `URLCanonicalizer` class needs to be called from several different packages, not just its own - so it has to be a public class - it can't be package-private and so this means that it's accessible to anyone.

The solution to this very general problem was to make a one-time change to Java's model of access control. This change applies both to user code calling the JDK and to applications calling third-party libraries.

2.1.4 New access control semantics

Modules add a new concept to Java's access control model - the idea of *exporting* a package. In Java 8 and before, code in any package can call public methods on any public class in any package.

This is sometimes called "shotgun privacy", after this famous quote about another programming language:

"Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun" — Larry Wall

For Java, however, shotgun privacy represented a major problem. More and more libraries were using internal APIs to provide capabilities that were difficult or impossible to provide in another way, and this threatened to harm the long-term health of the platform.

As of Java 8 there was no way to enforce access control across an entire package. This meant that the JDK team were unable to define a public API and know with certainty that clients of that API could not subvert it or directly link to the internal implementation.

The convention that anything in a package that starts `java` or `javax` is a public API and everything else is internal-only is just that - a convention. There is no VM or classloading mechanism that enforces that, as we've already seen.

With modules, however, this changes. The `exports` keyword has been introduced to indicate which packages are considered the public API of a module.

In the modular JDK the package `sun.net` is not exported - so the above Java 8 URL canonicalizer code will not compile. Here's what happens when we try with Java 11:

```
$ javac src/ch02/MyURLHandler.java
src/ch02/MyURLHandler.java:3: error: package sun.net is not visible
import sun.net.URLCanonicalizer;
 ^
(package sun.net is declared in module java.base, which does not export
it to the unnamed module)
src/ch02/MyURLHandler.java:8: error: cannot find symbol
    return isSimpleHostName(url);
 ^
symbol:   method isSimpleHostName(String)
location: class MyURLHandler
2 errors
```

Note that the form of the error message explicitly says that the `sun.net` package is now not visible - the compiler cannot even see the symbol.

This is a fundamental change in the way Java access control works. Only methods on exported packages are accessible - it is no longer the case that a public method on a public class is automatically visible to all code everywhere.

However, this change may not be all that visible to many developers. If you're a Java developer who plays by the rules, you will never have called an API in an internal package directly. However, you might well use a library or a framework that does, so it's good to understand what has actually changed, and avoid the FUD.

NOTE

Proper encapsulation is not free, and pre-modular Java is actually a very open system. It is perhaps only natural that when confronted with the more structured system that modules provide, many Java developers find some of the extra protections constraining or frustrating.

Let's meet the syntax that encodes these new semantics of Java modules.

2.2 Basic modules syntax

A Java Platform Module is defined as a conceptual unit, which is a collection of packages and classes that are declared and loaded as a single entity.

Each module must declare a new file, called a *module descriptor* - represented as a `module-info.java` file which contains:

- Module Name
- Module Dependencies
- Public API (Packages Exported)
- Reflective Access Permissions
- Services Provided
- Services Consumed

This file must be placed in a suitable place within the source hierarchy. For example, within a Maven style layout, the full module name `com.wellgrounded` appears directly after `src/main/java` and contains `module-info.java` and the package root.

```
src
main
java
    com.wellgrounded
        com
            wellgrounded
                VMIntrospector.java
                Discovery.java
        module-info.java
```

This is, of course, slightly different from non-modular Java projects, which often nominate `src/main/java` as the root of the source directories. However, the familiar hierarchical structure of packages under the module root is still visible.

NOTE

When a modular project is built, the module descriptor will be compiled into a class file `module-info.class` - but that file (despite its name) is actually quite different from the usual sort of class file that we see in the Java platform.

In this chapter we will address the basic directives of the descriptor but will not be delving deeply into all of the capabilities that modules provide. In particular, we will not discuss the services aspects of modules.

A simple example of a module descriptor looks like this:

```
module wgjd.discovery {
    exports wgjd.discovery;

    requires java.instrument;
    requires jdk.attach;
    requires jdk.internal.jvmsstat;
}
```

This contains 3 new keywords - `module`, `exports`, and `requires` in a syntax that should be suggestive to most Java programmers. The keyword `module` simply declares the opening scope of the declaration.

NOTE

The name `module-info.java` is reminiscent of `package-info.java`, and they are somewhat related. As packages are not really visible to the runtime, a workaround (hack?) was needed to provide a hook for annotation metadata that was intended to apply to the whole package. This hack was `package-info.java`. In the modular world, much more metadata can be associated with a module, and so a similar name was chosen.

The new syntax actually consists of *restricted keywords* - which are described in the Java

Language Specification like this:

A further ten character sequences are restricted keywords: open, module, requires, transitive, exports, opens, to, uses, provides, and with. These character sequences are tokenized as keywords solely where they appear as terminals in the ModuleDeclaration and ModuleDirective productions.

In simpler language this means that these new keywords will only appear in the descriptor for the module metadata and are not treated as keywords in general Java source.

However, it is good practice to avoid these words as Java identifiers, even if it is technically legal to use them. This is the same situation as we saw with var in Chapter 1, and we will use looser language and refer to them as "keywords" throughout the rest of the book.

2.2.1 Exporting and requiring

The exports keyword expects an argument, which is a package name. In our example:

```
exports wgjd.discovery;
```

means that our example discovery module exports the package wgjd.discovery - but as the descriptor does not mention any other packages then wgjd.discovery.internal is not exported and is not normally available to code outside the discovery module.

Multiple exports lines are possible in a module descriptor and in fact are quite usual. Fine-grained control is also possible with the exports ... to ... syntax that indicates only certain external modules may access a specified package from this module.

NOTE

A single module exports one or more packages that constitute the public API of the module and which are the only packages that code in other modules may access (usually).

The requires keyword declares a dependency of the current module and always requires an argument, which is a *module* name, rather than a package name.

The java.base module contains the most fundamental packages and classes of the Java runtime. We can use the jmod command to take a look:

```
$ jmod describe $JAVA_HOME/jmods/java.base.jmod
java.base@11.0.3
exports java.io
exports java.lang
exports java.lang.annotation
exports java.lang.invoke
exports java.lang.module
exports java.lang.ref
exports java.lang.reflect
exports java.math
exports java.net
exports java.net.spi
exports java.nio
// ... many, many more lines of output
```

These packages are used by every Java program, and so `java.base` is always an implicit dependency of every module, so does not need to be explicitly declared in `module-info.java`. This is in much the same way that `java.lang` is an implicit `import` into every Java class.

Some of the basic rules and conventions for module names include:

- Modules live in a global namespace
- Module names must be unique
- Use the standard `com.company.project` convention if appropriate

One important basic modules concept is *transitivity*. Let's take a closer look at this concept, as it occurs not only in the context of modules but also in Java's more familiar library (i.e. jar file) dependencies (which we will meet in Chapter 11).

2.2.2 Transitivity

Transitivity is a very general computing term, not specific to Java at all, which describes the situation that occurs when a code unit requires other units to function correctly, and those units can themselves require other units. Our original code may never even mention these "one step removed" code units - but they still need to be present or our application will not work.

To understand why this is the case, and why it is important, consider two modules `A` and `B` where `A` requires `B`. There are two cases:

1. `A` does not export any methods that mention types from `B` directly
2. `A` includes types from `B` as part of its API

In the case where `A` exports methods that return types that are defined in `B`, then this would have the effect that `A` is not usable unless clients of `A` (those modules that require `A`) also require `B`. This is quite an unnecessary overhead on clients of `A`.

The modules system provides some simple syntax to solve this: `requires transitive`. If a module `A` requires another module transitively, then any code that depends on `A` will also, implicitly, pick up the transitive dependencies as well.

Whilst usage of requires transitive is unavoidable in some use cases, in general, when writing modules minimizing use of transitivity is considered a best practice. We will have more to say about transitive dependencies when we discuss build tools in Chapter 11.

2.3 Loading modules

If the first time you've encountered Java classloading is when we briefly mentioned it in Chapter 1 and you have no other experience of it, then don't worry. The most important thing to know right now is that there are four different types of modules, some of which have slightly different behaviors when loaded:

- Platform Modules
- Application Modules
- Automatic Modules
- Unnamed Module

On the other hand, if you're already familiar with classloading, then you should know that the arrival of modules has changed some of the details of the way that classloading operates. A modern JVM has module-aware classloaders and the way that the JRE classes are loaded is quite different than in Java 8.

NOTE

We will meet classloading properly in Chapter 4, and introduce the modern way of doing things for both new and experienced readers.

The fundamental principles of the modular approach to classloading are:

- Modules are resolved from the module path, and not the old-school classpath.
- At startup the JVM resolves a graph of modules, which must be acyclic.
- One module is the root of the graph and is where execution starts from - it contains the class with the main method that will be the entry point.

Dependencies that have already been modularized are known as *application modules* and are placed on the module path. Unmodularized dependencies are placed on the familiar classpath, and are co-opted into the modules system via a migration mechanism.

Module resolution uses depth-first traversal, and because the graph must be acyclic, the resolution algorithm will terminate (and in linear time).

Let's delve a little more deeply into each of the four types of modules.

2.3.1 Platform modules

These are modules from the modular JDK itself - they would have been part of the monolithic runtime (`rt.jar`) in Java 8 (or possibly ancillary jars, such as `tools.jar`).

We can get a list of the available platform modules from the `--list-modules` flag:

```
$ java --list-modules
java.base@11.0.6
java.compiler@11.0.6
...
java.xml@11.0.6
java.xml.crypto@11.0.6
jdk.accessibility@11.0.6
...
jdk.unsupported@11.0.6
...
```

This will give an unabridged list, rather than the partial set that we saw in Figure 1.1

NOTE

The exact list of modules and their names will depend on the version of Java in use. For example, on Oracle's GraalVM implementation some additional modules like `com.oracle.graal.graal_enterprise`, `org.graalvm.js.scriptengine` and `org.graalvm.sdk` may be present.

The platform modules make heavy use of the *qualified exporting* mechanism wherein some packages are only exported to a specified list of modules and are not made generally available.

The most important module in the distribution is `java.base` - which is always an implicit dependency of every other module. It contains `java.lang`, `java.util`, `java.io` and various other basic packages. The module basically corresponds to the smallest possible Java runtime than an application could require and still run.

At the other end of the spectrum are the *aggregator modules* that don't contain any code but which serve as a shortcut mechanism to allow applications to bring in a very broad set of dependencies transitively. For example, the `java.se` module brings in the entire Java SE platform.

2.3.2 Application modules

These type of modules are the modularized dependencies of an application, or the application itself. This type of module is also sometimes known as a *library module*.

NOTE

There is no technical distinction between platform and application modules - the difference is purely philosophical - and which classloader is used to load them, as we will discuss in Chapter 3.

The third-party libraries that an application depends on will be application modules. For example, the Jackson libraries for manipulating JSON have been modularized as of version 2.10 and count as application (aka library) modules.

Application modules will typically depend upon both platform modules and other application modules. It is a good idea to try to constrain the dependencies of these modules as much as possible and to avoid requiring, e.g., `java.se` as a dependency.

2.3.3 Automatic modules

One deliberate design feature of the modules system is that you can't reference the classpath from a module.

This restriction seems to be potentially problematic - what happens if a module needs to depend on some code that has not yet been modularized?

The solution is to move the non-modular JAR file onto the module path (and remove it from the classpath).

When this is done, the jar becomes an *automatic module*. The modules system will automatically generate a name for your module, which is derived from the jar's name.

It is even possible to explicitly declare a name, by adding an entry for `Automatic-Module-Name` into the `MANIFEST.MF` file in the jar. This is often done as an intermediate step when migrating to Java modules, as it allows developers to reserve a module name and start to gain some of the benefits of interoperating with modular code.

For example, the Apache Commons Lang library is not yet fully modularized, but it provides `org.apache.commons.lang3` as an automatic module name.

NOTE	Automatic modules do not have proper module dependency information, so they are not first class citizens in the modules system and do not provide the same level of guarantees as genuine Java modules.
-------------	--

An automatic module exports every package that it contains, and automatically depends upon all other modules in the module path.

2.3.4 Unnamed module

All classes and jars on classpath are added to a single module, which is the unnamed module, or UNNAMED. This helps to maintain backwards compatibility but at the cost that the modules system is not as effective as it might be all the time that some code remains in the unnamed module.

NOTE This means that until all an application's dependencies are modularized, then the benefits of the modules system for your application are limited.

For the case of non-modular apps (e.g. Java 8 apps that are running on top of a Java 11 runtime) then the contents of the classpath are dumped into the unnamed module, and the root module is taken to be `java.se`

Modular code cannot depend on the unnamed module, and so in practice this means that modules cannot depend upon anything in the classpath. Automatic modules are often used to help resolve this situation.

Formally, the unnamed module depends upon all modules in JDK and on the module path as it is replicating the pre-modular behavior.

2.4 Building a first modular app

Let's build a first example of a modular application. To do this, we need to build a module graph (which is, of course, a DAG).

The graph must have a *root module*, which in our case is the module containing the entry point class of the app. The module graph of the application is the transitive closure of all modular dependencies of the root module.

As our example, we're going to adapt the HTTP site-checking tool we created in Chapter 1 to become a modular app. The files will be laid out like this:

```
.
wgjd.sitecheck
  wgjd
    sitecheck
      concurrent
        ParallelHTTPChecker.java
      internal
        TrustEveryone.java
        HTTPChecker.java
        SiteCheck.java
      module-info.java
```

We're breaking out certain concerns (e.g. the `TrustEveryone` provider) into their own classes rather than representing them as static inner classes - as we had to when all the code needed to live in a single file. We've also set up separate packages and will not be exporting all of them.

The module file is very similar the same as the one we met earlier:

```
module wgjd.sitecheck {
    // requires java.net.http;
    exports wgjd.sitecheck;
    exports wgjd.sitecheck.concurrent;
}
```

However, in order to investigate what happens when a dependency is missed, we've commented out the dependency on the HTTP module.

```
$ javac -d out wgjd.sitecheck/module-info.java wgjd.sitecheck/wgjd/
    sitecheck/*.java wgjd.sitecheck/wgjd/sitecheck/*/*.java
wgjd.sitecheck/wgjd/sitecheck/SiteCheck.java:8: error:
    package java.net.http is not visible
    import java.net.http.*;
        ^
    (package java.net.http is declared in module java.net.http, but
     module wgjd.sitecheck does not read it)
wgjd.sitecheck/wgjd/sitecheck/concurrent/ParallelHTTPChecker.java:4:
    error: package java.net.http is not visible
    import java.net.http.*;
        ^
// Several similar errors
```

This failure shows that simple problems with modules can be very easy to solve - the modules system has detected the missing module and is trying to help by suggesting a solution - add the missing module as a dependency. If we make that change then, as expected, the module builds without complaint.

However, more complex problems may require changes to the compilation step or manual intervention via a switch to control the modules system.

2.4.1 Command-Line switches for modules

When compiling a module then there are a number of command-line switches that can be used to control the modular aspects of the compile (and, later, execution). The most-commonly-encountered of these switches are:

- `list-modules` – Prints a list of all modules
- `module-path` – Specify one or more dirs that contain your modules
- `add-reads` – Add an additional `requires` to the resolution
- `add-exports` – Add an additional `exports` to the compilation
- `add-opens` – Enables reflective access to all types at runtime
- `add-modules` – Adds the list of modules to the default set
- `illegal-access=permit|warn|deny` - Change reflective access rule

We have already met the majority of these concepts already - with the exception of the qualifiers related to reflection which we will discuss in detail in Section 2.4.3

Let's see one of these switches in action. This will demonstrate a common issue with module

packaging and serves as an example of a real-world issue that many developers may encounter when starting to use modules with their own code.

When starting to work with modules, we sometimes find that we need to break encapsulation. For example, an application that has been ported from Java 8 may be expecting to access an internal package that is no longer exported.

For example, let's consider a project with a simple structure that uses the *Attach API* to dynamically connect to other JVMs running on a host and report some basic information about them. It's laid out on disc like this, just as we saw in an earlier example.

```
.
wgjd.discovery
wgjd
discovery
internal
    AttachOutput.java
Discovery.java
VMIntrospector.java
module-info.java
```

Compiling the project gives a series of errors:

```
$ javac -d out/wgjd.discovery wgjd.discovery/module-info.java \
wgjd.discovery/wgjd/discovery/*.java \
wgjd.discovery/wgjd/discovery/internal/*
wgjd.discovery/wgjd/discovery/VMIntrospector.java:4: error: package
    sun.jvmstat.monitor is not visible
import sun.jvmstat.monitor.MonitorException;
^
(package sun.jvmstat.monitor is declared in module jdk.internal.jvmstat,
 which does not export it to module wgjd.discovery)
wgjd.discovery/wgjd/discovery/VMIntrospector.java:5: error: package
    sun.jvmstat.monitor is not visible
import sun.jvmstat.monitor.MonitoredHost;
^
(package sun.jvmstat.monitor is declared in module jdk.internal.jvmstat,
 which does not export it to module wgjd.discovery)
```

These problems are being caused by some code in the project that makes use of internal APIs:

```

public class VMIntrospector implements Consumer<VirtualMachineDescriptor> {

    @Override
    public void accept(VirtualMachineDescriptor vmd) {
        var isAttachable = false;
        var vmVersion = "";
        try {
            var vmId = new VmIdentifier(vmd.id());
            var monitoredHost = MonitoredHost.getMonitoredHost(vmId);
            var monitoredVm = monitoredHost.getMonitoredVm(vmId, -1);
            try {
                isAttachable = MonitoredVmUtil.isAttachable(monitoredVm);
                vmVersion = MonitoredVmUtil.vmVersion(monitoredVm);
            } finally {
                monitoredHost.detach(monitoredVm);
            }
        } catch (URISyntaxException | MonitorException e) {
            e.printStackTrace();
        }

        System.out.println(
            vmd.id() + '\t' + vmd.displayName() + '\t' + vmVersion +
            '\t' + isAttachable);
    }
}

```

While classes like `VirtualMachineDescriptor` are part of the exported interface of the `jdk.attach` module (as the class is in the exported package `com.sun.tools.attach`), other classes that we depend upon (such as `MonitoredVmUtil` in `sun.jvmstat.monitor`) are not accessible.

Fortunately, the tools provide a way to soften the module boundaries and provide access to a non-exported package.

To achieve this we need to add a switch `--add-exports` to force access to the internals of the `jdk.internal.jvmstat` module - which means we are definitely breaking encapsulation by doing this. The resulting compilation command line looks like this:

```
$ javac -d out/wgjd.discovery --add-exports=jdk.internal.jvmstat/
  sun.jvmstat.monitor=wgjd.discovery wgjd.discovery/module-info.java
  wgjd.discovery/wgjd/discovery/*.java wgjd.discovery/wgjd/discovery/
  internal/*
```

The syntax of `--add-exports` is that we must provide the module and package name that we require access to, and which module is being granted the access.

2.4.2 Executing a modular app

Up until the arrival of modules, there were only two methods of starting a Java application:

```
java -cp classes wgjd.Hello
java -jar my-app.jar
```

These should both be familiar to Java programmers as launching a class and the main class from within a JAR file. In modern Java two more methods of launching programs have been added.

We met a new way of launching single source-file programs in Section 1.5.4, and now we're going to meet the fourth mode - launching the main class of a module. The syntax is:

```
java --module-path mods -m my.module/my.module.Main
```

However, just as for compilation, we may need additional command-line switches. For example, for our earlier example of introspection:

```
$ java --module-path out -m wgjd.discovery/wgjd.discovery.Discovery
Exception in thread "main" java.lang.IllegalAccessException:
  class wgjd.discovery.VMIntrospector (in module wgjd.discovery) cannot
  access class sun.jvmstat.monitor.MonitorException (in module
  jdk.internal.jvmstat) because module jdk.internal.jvmstat does not
  export sun.jvmstat.monitor to module wgjd.discovery
at wgjd.discovery/wgjd.discovery.VMIntrospector.accept(
  VMIntrospector.java:19)
at wgjd.discovery/wgjd.discovery.Discovery.main(Discovery.java:26)
```

To prevent this error, we must also provide the encapsulation-breaking switch to the actual program execution:

```
$ java --module-path out --add-exports=jdk.internal.jvmstat/
  sun.jvmstat.monitor=wgjd.discovery -m wgjd.discovery/
  wgjd.discovery.Discovery
Java processes:
PID  Display Name      VM Version   Attachable
53407  wgjd.discovery/wgjd.discovery.Discovery    15-ea+24-1168     true
```

If the runtime system can't find the root module we asked for, then we expect to see an exception like this:

```
$ java --module-path mods -m wgjd.hello/wgjd.hello.HelloWorld
Error occurred during initialization of boot layer
java.lang.module.FindException: Module wgjd.hello not found
```

Even this simple error message is showing us that we have new aspects to the JDK, including:

- Packages - including `java.lang.module`
- Exceptions including `FindException`

showing once again that the modules system really has become an integral part of the execution of every Java program, even if it is not always immediately obvious.

In the next section, we'll briefly introduce the interaction of modules with reflection. We assume that you're already familiar with reflection - but if you're not then feel free to skip this section for now and come back to it after you've read Chapter 3, which contains an introduction to classloading and reflection.

2.4.3 Modules and reflection

In Java 8 developers can use reflection to access almost anything in the runtime. There's even a way to bypass the access control checks in Java and, for example, call private methods on other classes via the so-called " `setAccessible()` hack".

As we've already seen, modules change the rules for access control. This also applies to reflection - the intent is that by default only exported packages should be accessed reflectively.

However, the developers of the modules system realized that sometimes developers want to give reflective access (but not direct access) to certain packages. This requires an explicit permission and can be achieved by using the `opens` keyword to provide reflective-only access to an otherwise internal package.

Developers can also specify fine-grained access by using the syntax `opens ... to ...` to allow a named set of packages be opened reflectively to specific modules, but not more generally.

At this point it is good to reflect that many well-grounded developers may well have, buried somewhere in their code, a "gnarly hack" that uses reflection to get at something private or internal.

The above discussion seems to imply that these types of reflective tricks are now ruled out. The truth is a little more complicated, and is best explained via a discussion of the command-line switch `--illegal-access`. This switch comes with three settings `permit|warn|deny` and is used to control the strictness of checks on reflection.

The intent of the modules system is that over time the entire Java ecosystem moves towards proper encapsulation, including reflection and that in a future version of Java the switch will default to `deny` - instead of the current setting - `permit` - which has been in use since Java 9.

NOTE	This change obviously cannot happen overnight - if the reflection switch was suddenly set to <code>deny</code> then huge swathes of the Java ecosystem would break - and no-one would upgrade.
-------------	--

One additional useful concept to help this transition is *open modules*. This is a simple declaration that is used to allow for completely open reflective access - it opens all the module's packages for reflection but not compile-time access.

This provides simple compatibility with existing code & frameworks but is a looser form of encapsulation. For this reason, open modules are best avoided or used only as a transitional form when migrating to a modular build.

In Chapter 16 we will discuss the specific case of `Unsafe` which is a great example to indicate some of the problems with reflection in a modular world.

2.5 Architecting for modules

Modules represent a fundamentally new way of packaging and deploying code. Teams do need to adopt some new practices in order to get the most out of the new functionality and the architectural benefits. However, the good news is that there's no need to start doing that straightaway just to start using modern Java.

The traditional, old-school methods using the classpath and jar files will continue to work until such time as teams are ready to adopt modules wholeheartedly.

In fact, Mark Reinhold (Chief Architect for Java at Oracle) had this to say about the "need" for applications to adopt modularity.¹

"There is no need to switch to modules.

There has never been a need to switch to modules.

Java 9 and later releases support traditional JAR files on the traditional class path, via the concept of the unnamed module, and will likely do so until the heat death of the universe.

Whether to start using modules is entirely up to you.

If you maintain a large legacy project that isn't changing very much, then it's probably not worth the effort."

In an ideal world, modules would be the default for all greenfield apps, but this is proving to be complex in practice, so as an alternative, when migrating a process like this can be followed:

1. Upgrade to Java 11 (classpath only)
2. Set an automatic module name
3. Introduce a *monolithic module* consisting of all code
4. Break out into individual modules as needed

Typically, at step 3, there is way too much implementation code exposed. This means that quite often part of the work of step 4 is to create additional packages to house internal and implementation code and to refactor code into them.

If you are still using Java 8 and you aren't ready yet to migrate to a modular build, then there are still several things that you can do to prepare your code for the migration:

- Introduce an automatic module name in `MANIFEST.MF`
- Remove Split Packages from your deployment artifacts
- Use `jdeps` and Compact Profiles to reduce your footprint of unnecessary dependencies

To take the first of these - the use of an explicit automatic module name (as we discussed earlier in the chapter) will ease the transition. The automatic module name will be ignored by all versions of Java that do not support modules, but will still allow you to reserve a stable name for your library, and to move some code out of the unnamed module.

It also has the advantage that consumers of your library are prepared for the transition to modules - as you have already advertised the name the module will be using.

Let's take a closer look at the other two concrete recommendations.

2.5.1 Split packages

One common problem that developers encounter when they start to use modules is *split packages*. This is the situation in which two or more separate jars contain classes belonging to the same package.

In a non-modular application there is no problem with split packages - because neither jar files nor packages have any particular significance to the runtime.

However, in the modular world, a package must belong to only one module and cannot be split.

If an existing application is upgraded to use modules and has dependencies that contain split packages then this will have to be remediated - there just is no way around it.

For code that the team controls, this is additional work, but not too difficult. One technique is to have a specific artifact (often using a `-all` suffix) that is generated by the build system alongside the non-modular versions, with a single jar containing all parts of the split package.

For external dependencies, this can be more complicated. It may be necessary to repackage third-party open source code into a jar that can be consumed as an automatic module.

One other useful stepping stone to modules is "Compact Profiles", which were introduced in Java 8 as a small hint of the modularity story that would arrive in Java 9.

2.5.2 Java 8 compact profiles

Compact Profiles are a Java 8 feature. They are runtime environments that are reduced in size, and which must implement both the JVM and Java language specifications.

NOTE	Profiles must include all classes and packages which are explicitly mentioned in the Java language specification.
-------------	---

Profiles are lists of packages, and they are usually identical to the package of the same name in the full Java SE platform. There are a very few exceptions to this but they are explicitly called out.

One of the main use cases of Profiles is as the basis for a server application or other environment, where deploying unnecessary capabilities is undesirable.

For example, historically, a large number of security vulnerabilities were connected to Java's GUI features, especially in Swing and AWT. By choosing not to deploy the packages which implement those features in applications where they are not needed, we can gain a modest amount of additional security, especially for, e.g. server applications.

Compact 1 is the smallest set of packages which it is feasible to deploy an application on. It contains 50 packages, from the very familiar:

- java.io
- java.lang
- java.math
- java.net
- java.text
- java.util

to some perhaps more unexpected packages that nonetheless provide essential classes to modern applications:

- java.util.concurrent.atomic
- java.util.function
- javax.crypto.interfaces
- javax.net.ssl
- javax.security.auth.x500

Compact 2 is significantly larger, containing packages such as those needed for XML, SQL, RMI and security. Compact 3 is larger still, and basically consists of the entire JRE, minus the windowing and GUI components - similar to the `java.se` module.

NOTE All profiles ship the transitive closure of types referred to by Object and all types mentioned within the language specification.

The Compact 1 profile is the closest to a minimal runtime - so in some ways it resembles a prototypical form of the `java.base` module.

NOTE Ideally, if your application or library can be made to run with only Compact 1 as a dependency, then it should.

To help determine whether your app can run with Compact 1, or another profile, the JDK

provides `jdeps`. This is a static analysis tool that ships with Java 8 & 11 for examining the dependencies of packages or classes.

The tool can be used in a number of different ways, from identifying which profile an application needs to run under, to identifying developer code that makes calls into the undocumented, internal JDK APIs (such as the `sun.misc` classes), through to helping trace transitive dependencies.

It can be very helpful for migrations from Java 8 to 11, and works with both JARs and modules. In its simplest form, `jdeps` takes a class or package and provides a brief list of packages that are the dependencies. For example, for the discovery example:

```
$ jdeps Discovery.class
Discovery.class -> java.base
Discovery.class -> jdk.attach
Discovery.class -> not found
    wgjd.discovery      -> com.sun.tools.attach      jdk.attach
    wgjd.discovery      -> java.io                  java.base
    wgjd.discovery      -> java.lang                java.base
    wgjd.discovery      -> java.util                java.base
    wgjd.discovery      -> wgjd.discovery.internal not found
```

The `-P` switch shows which profile is needed for a class (or package) to run, although of course this only works for a Java 8 runtime.

Let's move on to take a quick look at another migration technique that a well-grounded Java developer should be aware of - the use of *multi-release jars*.

2.5.3 Multi-release JARs

This new capability allows the construction of a jar file that can house libraries and components that can work on both Java 8 and modern, modular JVMs.

This means that you can, for example, depend on library classes that are only available in later versions but can still run on an earlier version by using a fallback and stubbing approach.

To make a multi-release jar, an entry must be included in the jar's manifest file:

```
Multi-Release: True
```

This entry is only meaningful to JVMs from version 9 upwards, which means that if the jar is used on a Java 8 (or earlier) VM then the multi-release nature will be ignored.

The classes that target post-8 versions are referred to as *variant code* and are stored in a special directory in `META-INF` within the jar:

```
META-INF/versions/<version number>
```

The mechanism works by overriding on a per-class basis. Versions 9 and upwards of Java will look for a class that has the exact same name in the `versions` directories as in the main content root. If one is found, then the overridden version is used in place of the class in the content root.

NOTE

Java class files are stamped with the version number of the Java compiler that created them - the classfile version number - and code created on a later Java release will not run on an older JVM.

The `META-INF/versions` location is ignored by Java 8 and below, so this provides a clever trick to sidestep the fact that some of the code that is contained in a multi-release jar has too high a classfile version to run on Java 8.

However, this does mean that the API of the class in the content root and any overridden variants must be the same, as they will be linked in exactly the same way for both cases.

EXAMPLE: BUILDING A MULTI-RELEASE JAR

Let's look at providing an example capability - getting the process ID of the running JVM. Unfortunately, this is somewhat cumbersome on versions of Java before 9, and requires some low-level hackery in the `java.lang.management` package.

Java 11 does provide an API for getting a PID from the Process API, and so we want to set up a simple multi-release JAR which will use the simpler API when it is available and fall back to the JMX based approach only if necessary.

The main class looks like this:

```
public class Main {
    public static void main(String[] args) {
        System.out.println(GetPID.getPid());
    }
}
```

and note that the capability that has a version-dependent implementation has been isolated into a separate class `GetPID`. The Java 8 version of the code is somewhat verbose:

```

public class GetPID {
    public static long getPid() {
        System.out.println("Java 8 version..."); ①
        // ManagementFactory.getRuntimeMXBean().getName() returns the name that
        // represents the currently running JVM. On Sun and Oracle JVMs, this
        // name is in the format <pid>@<hostname>.

        final var jvmName = ManagementFactory.getRuntimeMXBean().getName();
        final var index = jvmName.indexOf('@');
        if (index < 1) {
            return 0;
        }

        try {
            return Long.parseLong(jvmName.substring(0, index));
        } catch (NumberFormatException e) {
            return 0;
        }
    }
}

```

- ① We include this line so we can see that this is the Java 8 version

This requires us to parse a string from a JMX method - and even then our solution is not guaranteed to be portable across JVM implementations.

By contrast, Java 9 and upwards provides a much simpler standard method in the API:

```

public class GetPID {
    public static long getPid() {
        // Use the ProcessHandle API, new in Java 9 ...
        var ph = ProcessHandle.current();
        return ph.pid();
    }
}

```

and as the `ProcessHandle` class is in the package `java.lang` - so we don't even need an import statement.

We now need to arrange the multi-release jar so that the Java 11 code is included in the jar, and is used in preference to the fallback version, if the JVM has a high enough version.

One suitable code layout looks like this:

```

.
src
  main
    java
      wgjd2e
        Main.java
        GetPID.java
  versions
    java
      11
        wgjd2e
          GetPID.java

```

The main part of the codebase needs to be compiled with Java 8, and then the post-8 code is

compiled afterwards with a different Java version, before being package into a multi-release jar "by hand" (i.e. using the command-line `jar` tool directly).

NOTE This code layout uses the convention that the Maven and Gradle build tools follow, which we'll meet properly in Chapter 11.

Let's compile the code from the command-line using `javac` (version 8):

```
$ javac -sourcepath src/main/java/ -d out src/main/java/wgjd2e/*.java
```

Next, we need to change JDK to version 11 and then set up the directory for the variant code `out-11`, and then compile the variant code:

```
$ javac -sourcepath versions/11 -d out-11 versions/11/wgjd2e/GetPID.java
```

We also need a `MANIFEST.MF` file, but we can use the (Java 11) `jar` tool to automatically construct what we need:

```
$ jar --create --file pid.jar --main-class=wgjd2e.Main -C out/ .
--release 11 -C out-11/ .
```

This creates a multi-release jar which is also runnable (with `Main` being the entry point class). Running the jar gives this output on Java 11:

```
$ java -version
openjdk version "11.0.3" 2019-04-16
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.3+7)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.3+7, mixed mode)

$ java -jar pid.jar
13855
```

and under Java 8:

```
$ java -version
openjdk version "1.8.0_212"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_212-b03)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.212-b03, mixed mode)

$ java -jar pid.jar
Java 8 version...
13860
```

Note the extra banner we added to the Java 8 version so you can distinguish between the two cases and be sure that the two different classes are actually being run. For real-world use cases for multi-release jars, we would want the code to either perform the same in both cases (if we're shimming a capability back to Java 8) or to fail in a graceful or predictable way if run on a JVM that doesn't support a capability.

One important architectural pattern that we recommend following is isolating JDK

version-specific code into a package or group of packages, depending on how extensive the capability is.

Some basic guidelines and principles for the project include:

- The main codebase must be able to be built with Java 8
- Java 11 portion must be built with Java 11
- Java 11 portion must be in a separate code root - isolated from the main build
- The end result should be a single JAR
- Keep the build configuration as simple as possible
- Consider making the multi-release jar modular as well

The last point is especially important, and this continues to be true for more complex projects that will inevitably end up requiring a proper build tool, rather than just `javac` and `jar`.

2.6 Beyond modules

To conclude the chapter, let's take a quick look at what lies beyond modules.

Recall that the entire point of modules was to introduce a missing abstraction into the Java language - the idea of deployment units with dependency guarantees that can be relied upon by the source code compiler and runtime.

This idea of modular dependency information that can be trusted has a number of applications to the modern world of deployable software. We will meet other examples towards the end of the book, but let's conclude this chapter by meeting a new capability that was added to the platform along with modules - *JLink*.

This is the ability to package a reduced Java runtime along with an application. This provides several benefits to applications that make use of it:

- Package application and JVM into a single self-contained directory
- Reduce the footprint and overall download size of the application and JRE bundle
- Reduce support overhead - as there is no need to debug interactions between a Java application and a host-installed JVM

The self-contained directories that `jlink` produces can easily be packaged as deployable artifacts (such as a Linux `.rpm` or `.deb`, a Mac `.dmg` or a Windows `.msi`), providing a simple installation experience for modern Java applications.

In some ways, the Compact Profiles technology in Java 8 provides an early version of JLink, but the version that arrived with modules is far more useful and comprehensive.

As an example, we're going to reuse the discovery example from earlier in the chapter. This has a simple `module-info.java`:

```
module wgjd.discovery {
    exports wgjd.discovery;

    requires java.instrument;
    requires java.logging;
    requires jdk.attach;
    requires jdk.internal.jvmstat;
}
```

This can be built into a JLink bundle via a command like this:

```
$ jlink --module-path $JAVA_HOME/jmods/:out --output bundle/ --add-modules
wgjd.discovery
```

In our simple example, we have produced a JLink bundle that can be delivered as a tar ball, or packaged into a Linux package (such as .deb or .rpm). We can actually take this one step further and use *static compilation* to convert such a bundle into a native executable - and we will discuss how this is achieved in Chapter 16.

We should sound a word of caution. JLink is a great piece of technology, but it has one important limitation that you should be aware of:

- It will only work with an application with fully modularized dependencies
- It does not work with non-modular code
- Even automatic modules are not sufficient.

This is because to be absolutely sure that all of the necessary parts of the JRE are included in the bundle, JLink relies upon the strongly declarative information in the module graph - and so requires a `module-info.class` for each dependency. Without this information building a reduced JRE is very likely to be unsafe.

Unfortunately, in the real world, many libraries that applications depend upon are still not fully modularized. This sharply reduces the usefulness of JLink. To solve this problem, toolmakers have developed plugins to repackage and synthesize "true" modules from unmodularized libraries.

However, to use these tools requires the use of a build system. This means we will defer real-world examples of JLink until later on, when we meet the build tools in Chapter 10.

2.7 Summary

Modules are a brand new concept in Java - one which should really have existed all along. They are not packages (although they group classes together). They are not old-school JAR archives (although they are sometimes packaged as `.jar` files).

Instead, modules group packages together and provide metadata about the module as an entire unit. The module declares its dependencies and public interfaces in a way that is enforced by the

compiler and runtime, and provides for fundamentally better software design for the coming decades.

Having said that, the benefits of modules come at a price - especially for legacy monolithic applications. The truth is that the extent to which modules have been adopted by the Java ecosystem is still patchy and incomplete, even 3 years after the first modular runtime was released. It is fair to assume that not every Java project (or developer) will ever adopt modules.

However, for projects that choose to make the transition there are a number of benefits:

- Modules allow library developers to have fine-grained control over what classes and methods are visible to their users.
- New syntax expressed in the `module-info.java` file controls how classes and methods are exposed within the module system.
- Modules require changes in classloading including making them aware of the restrictions a module defines and handling for loading non-modular code.
- Building with modules requires new command-line flags and a change to the standard layout for a Java project you may be aware of.
- Taking advantage of modules needs thought and revision for existing projects, particularly those where a single package is split between multiple JAR files.

At a high level, adopting modules allows projects to leverage these major benefits:

- Modules are a fundamentally better way to architect applications for modern deployments
- Modules are key to reducing footprint, especially in containers
- Modules pave the way for other new capabilities (such as static compilation)

Even for projects that will not cross the chasm, there are useful tools, such as Multi-release jars and Compact Profiles that can be used to help prepare an existing Java 8 projects to integrate with the modular ecosystem that the JVM has now become.

In the next chapter, we're going to meet the new language features that have arrived between Java 11 and 17 and get you ready to start coding in a Java 17 style.

Class files and bytecode

This chapter covers

- Classloading
- Reflection
- The anatomy of class files
- JVM bytecode and why it matters

One tried-and-trusted way to become a more well-grounded Java developer is to improve your understanding of how the platform works. Getting to grips with core features such as classloading and the nature of JVM bytecode can greatly help with this goal.

Consider the following scenarios that a senior Java developer might encounter:

Imagine you have an application that makes heavy use of Dependency Injection (DI) techniques such as Spring, and it develops problems starting up and fails with a cryptic error message. If the problem is more than a simple configuration error, you may need to understand how the DI framework is implemented to track down the problem. This means understanding classloading.

Or suppose that a vendor you're dealing with goes out of business — you're left with a final drop of compiled code, no source code, and patchy documentation. How can you explore the compiled code and see what it contains?

All but the simplest applications can fail with a `ClassNotFoundException` or `NoClassDefFoundError`, but many developers don't know what these are, what the difference is between them, or even why they occur.

This chapter focuses on the aspects of the platform that underlie these concerns. We'll also discuss some more advanced features — but they are intended for those folks that like to dive

deep and they can be skipped if you're in a hurry.

We'll get started with an overview of classloading — the process by which the VM locates and activates a new type for use in a running program. Central to that discussion are the `Class` objects that represent types in the VM. Next, we'll look at how this builds into the major language feature known as reflection.

After that, we'll discuss tools for examining and dissecting class files. We'll use `javap`, which ships with the JDK, as our reference tool. Following this class file anatomy lesson, we'll turn to bytecode. We'll cover the major families of JVM opcodes and look at how the runtime operates at a low level.

Let's get started by discussing classloading — the process by which new classes are incorporated into a running JVM process. In this section, we will first discuss the basics of "classic" classloading, as it was done in Java 8 and before. Later on in the chapter we will talk about how the arrival of the modular environment introduces some (small) changes to classloading.

4.1 Classloading and class objects

A `.class` file defines a type for the JVM, complete with fields, methods, inheritance information, annotations, and other metadata. The class file format is well-described by the standards, and any language that wants to run on the JVM must adhere to it.

NOTE

The class is the fundamental unit of program code that the Java platform understands and will accept and execute.

From the perspective of a beginning Java developer, a lot of the classloading mechanism is hidden from view. The developer provides the name of the main application class (which must be present on the classpath) and the JVM finds and executes the class.

Any application dependencies (e.g. libraries other than the JDK) must also be on the classpath, and the JVM finds and loads them as well. However, the Java specifications do not say whether this needs to be done at application startup, or later, as needed.

NOTE

The API that the Java classloading system presents to the user is fairly simple - a lot of the complexity is hidden on purpose, and we will discuss the developer-available API later in the chapter.

Let's start with a very simple example:

```
Class<?> clazz = Class.forName("MyClass");
```

This piece of code will load a class, `MyClass` into the current execution state. From the JVM's

perspective, to achieve this a number of steps must be performed. First, a class file corresponding to the name `MyClass` must be found and then the class it contains it must be *resolved*. These steps are performed in native code - in HotSpot the native method is called `JVM_DefineClass()`.

The actual process, at a high level, is that the native code builds the JVM's internal representation (which is called a *klass* and which we will meet properly in Chapter 15). Then, provided the *klass* could be extracted from the class file successfully, the JVM will construct a *Java mirror* of the *klass* - which is passed back to Java code as a `Class` object.

After this, the `Class` object representing the type is available to the running system, and new instances of it can be created.

NOTE The same process is used for the main application class and for all of its dependencies and any other classes that may be required after the program has started.

In this section, we'll cover the steps from the JVM's point of view in a bit more detail and provide an introduction to classloaders, which are the objects that control this entire process.

4.1.1 Loading and linking

One way of looking at the JVM is that it is an execution container. In this view, the purpose of the JVM is to consume class files and execute the bytecode they contain. To achieve this, the JVM must retrieve the contents of the class file as a data stream of bytes, convert it to a useable form, and add it to the running state - this is essentially the process that we are describing here.

This is a somewhat complex process that can be divided in a number of ways, but we refer to it as *loading* and *linking*.

NOTE Our discussion of loading and linking refers to some details that are specific to the HotSpot code but other implementations should do similar things.

The first step is to acquire the data stream of bytes that constitute the class file. This process starts with a byte array that is often read in from a filesystem (but other alternatives are definitely possible).

Once we have the stream, it must be parsed to check that it contains a valid class file structure. If so, then a candidate *klass* is created. During this phase, whilst the candidate *klass* is being filled in, some basic checks are performed (e.g. can the class being loaded actually access its declared superclass? does it try to override any `final` methods?)

However, at the end of the loading process, the data structure corresponding to the class isn't usable by other code yet and in particular we don't have a fully-functional klass (or `Class` object).

To get there, the class must now be linked. Logically speaking, this step breaks down into three subphases — verification, preparation, and resolution. However, in a real implementation, the code may not be cleanly separated out, so if you are planning to read the source code, you should be aware that the description provided here is a high-level or conceptual description of the process and does not have a precise correlation to the actual implementing code.

With this in mind, verification can be understood to be the sub-phase that confirms that the class conforms to the requirements of the Java specifications and won't cause runtime errors or other problems for the running system.

This relationship between the phases of linking can be seen in figure 4.1.

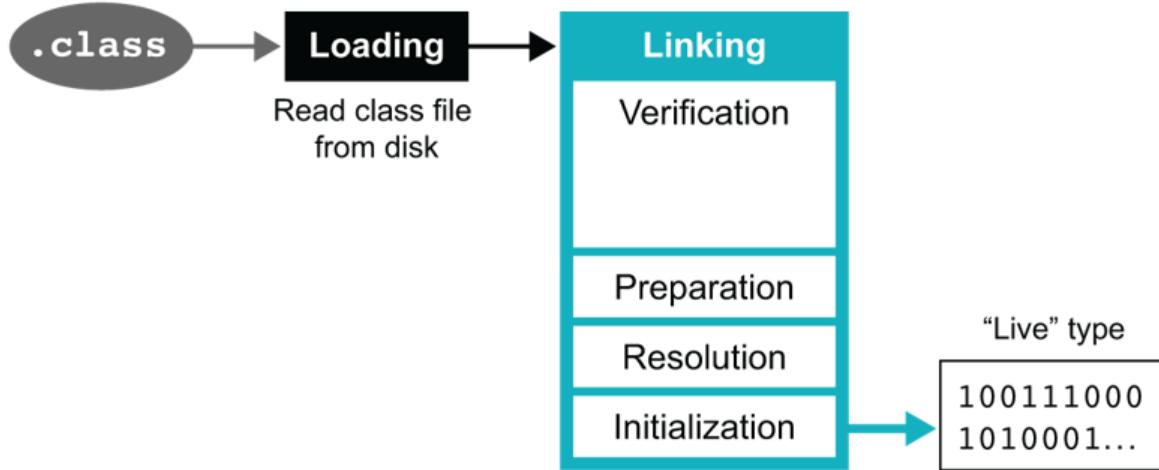


Figure 4.1 Loading and linking (with subphases of linking)

Let's meet each phase in turn.

VERIFICATION

Verification is quite a complex process, consisting of several independent concerns.

For example, the JVM needs to check symbolic information contained in the constant pool (discussed in detail in section 4.3.3) is self-consistent and obeys the basic behavior rules for constants.

Another major concern, and probably the most complex part of verification, is checking the bytecode of methods. This involves ensuring that the bytecode is well-behaved and doesn't try to circumvent the VM's environmental controls.

Some of the main checks that are performed include:

- Make sure bytecode doesn't try to manipulate the stack in disallowed or evil ways
- Make sure every branch instruction (e.g. from an `if` or a loop) has a proper destination instruction
- Check methods are called with the right number of parameters of the correct static types
- Check local variables are only assigned suitably typed values
- Check each exception that can be thrown has a legal catch handler

These checks are done for several reasons - including performance. The checks enable the skipping of runtime checks, thus making the interpreted code run faster. Some of them can also simplify the compilation of bytecode into machine code at runtime (just-in-time compilation, which we'll cover in Chapter 6).

PREPARATION

Preparing the class involves allocating memory and getting static variables in the class ready to be initialized, but it doesn't initialize variables or execute any VM bytecode.

RESOLUTION

Resolution is the part of linking where the VM checks that the supertype of the class being linked (and any interfaces that it implements) are already linked, and if they have not, then they are linked before the linking of this class continues. This can lead to a recursive linking process for any new types that have now been seen.

NOTE

A key phrase that relates to this aspect of classloading is the transitive closure of types. It means that not only the types that a class inherits from directly, but also all types that are indirectly referenced must be linked.

Once all additional types that need to be loaded have been located and resolved, the VM can initialize the class it was originally asked to load.

INITIALIZATION

In this final phase, any static variables can be initialized and any static initialization blocks run. This is a very significant point as it is only now that the JVM is finally running bytecode from the newly loaded class.

When this step completes, the class is fully loaded and ready to go. The class is available to the runtime and new instances of it can be created.

Any further classloading operations that refer to this class will now see that it is loaded and available.

4.1.2 Class objects

The end result of the linking and loading process is a `Class` object, which represents the newly loaded and linked type. It's now fully functional in the VM, although for performance reasons some aspects of the `Class` object are only initialized on demand.

NOTE

`Class` objects are regular Java objects. They live in the Java heap just like any other object.

Your code can now go ahead and use the new type and create new instances. In addition, the `Class` object of a type provides a number of useful methods, such as `getSuperClass()`, which returns the `Class` object corresponding to the supertype.

`Class` objects can be used with the *Reflection API* for indirect access to methods, fields, constructors, and so forth. A `Class` object has references to `Method` and `Field` objects that correspond to the members of the class. These objects can be used in the Reflection API to provide indirect access to the capabilities of the class, as we will see later on in this chapter.

You can see the high-level structure of this in figure 4.2.

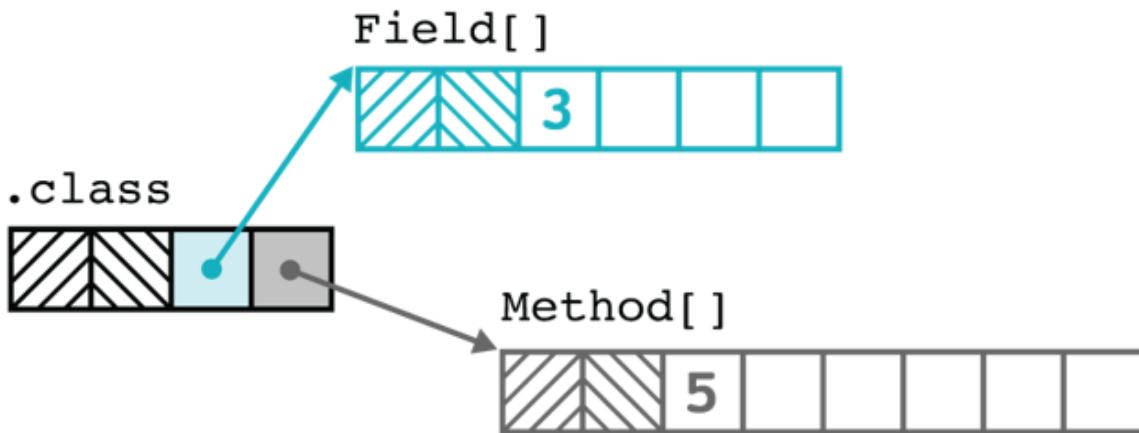


Figure 4.2 Class object and Method references

So far, we haven't discussed exactly which part of the runtime is responsible for locating and linking the byte stream that will become the newly loaded class. This is handled by classloaders — subclasses of the abstract class `ClassLoader`, and they're our next subject.

4.2 Classloaders

Java is a fundamentally object-oriented system with a dynamic runtime. One of the aspects of this is that Java's types are alive at runtime, and the type system of a running Java platform can be modified - in particular by the addition of new types.

This means that the types that make up a Java program are open to extension by unknown types at runtime (unless they are `final` or one of the new `sealed` classes). The classloading capability is exposed to the user - classloaders are just Java classes that extend `ClassLoader` - they are themselves Java types.

NOTE

In modern Java environments all classloaders are modular - loading classes is always done within the context of a module.

The class `ClassLoader` has some native methods, including the loading and linking aspects that are responsible for low-level parsing of the class file - but user classloaders are not able to override this aspect of classloading.

It is not possible to write a classloader using native code.

The platform ships with a number of typical classloaders, which are used to do different jobs during the startup and normal operation of the platform.

`BootstrapClassLoader` aka primordial classloader — This is instantiated very early in the process of starting up the VM - so it's usually best to think of it as being a part of the VM itself. It's typically used to get the absolute basic system loaded - essentially `java.base`.

`PlatformClassLoader` - After the bare minimum system has been bootstrapped then the platform classloader will load the rest of the platform modules that the application depends upon. This classloader is the primary interface to access any platform class - regardless of whether it was actually loaded by this loader or the bootstrap.

`AppClassLoader` - the application classloader — This is the most widely used classloader. This one will load the application classes and do the majority of the work in most modern Java environments. In modular environments the application class loader is no longer an instance of `URLClassLoader` (as it was in Java 8 and before) but, instead is an instance of an internal class.

Let's see these new classloaders in action, by adding a class called `DisplayClassloaders` to the `wgjd.discovery` module from Chapter 2:

```
package wgjd.discovery;

import com.sun.tools.attach.VirtualMachineDescriptor;

public class DisplayClassloaders {
    public static void main(String[] args) {
        var clThis = DisplayClassloaders.class.getClassLoader();
        System.out.println(clThis);
        var clObj = Object.class.getClassLoader();
        System.out.println(clObj);
        var clAttach = VirtualMachineDescriptor.class.getClassLoader();
        System.out.println(clAttach);
    }
}
```

This produces the output:

```
$ java --add-exports=jdk.internal.jvmstat/sun.jvmstat.monitor=wgjd.discovery
--module-path=out -m wgjd.discovery/wgjd.discovery.DisplayClassloaders
jdk.internal.loader.ClassLoaders$AppClassLoader@5fd0d5ae
null
jdk.internal.loader.ClassLoaders$AppClassLoader@5fd0d5ae
```

Notice that the classloader for `Object` (which is in `java.base`) reports as `null`. This is a security feature - the bootstrap classloader does no verification and provides full security access to every class it loads. For that reason it does not make sense to have the classloader represented and available within the Java runtime - too much potential for bugs and / or abuse.

In addition to their core role, classloaders are also often used to load resources (files that aren't classes, such as images or config files) from JAR files or other locations on the classpath.

This is often seen in a pattern that combines with try-with-resources to produce code like this:

```
try (var is = TestMain.class.getResourceAsStream("/resource.csv");
     var br = new BufferedReader(new InputStreamReader(is))) {
    // ...
}
// Exception handling elided
```

The classloaders provide this mechanism in a couple of different forms - returning either a `File` or an `InputStream` but unfortunately not a `Path`.

4.2.1 Custom classloading

In more complex environments, there will often be a number of additional *custom classloaders* - classes that subclass `java.lang.ClassLoader` (directly or indirectly). This is possible because the classloader class is not final, and developers are, in fact, encouraged to write their own classloaders that are specific to their individual needs.

NOTE

Custom classloaders are represented as Java types, so they need to be loaded by a classloader - which is usually referred to as their parent classloader. This should not be confused with class inheritance and parent classes - instead, classloaders are related by a form of delegation.

In figure 4.3, you can see the delegation hierarchy of classloaders, and how the different loaders relate to each other. In some special cases, a custom classloader may have a different classloader as their parent, but the usual case is that it is the loading classloader.

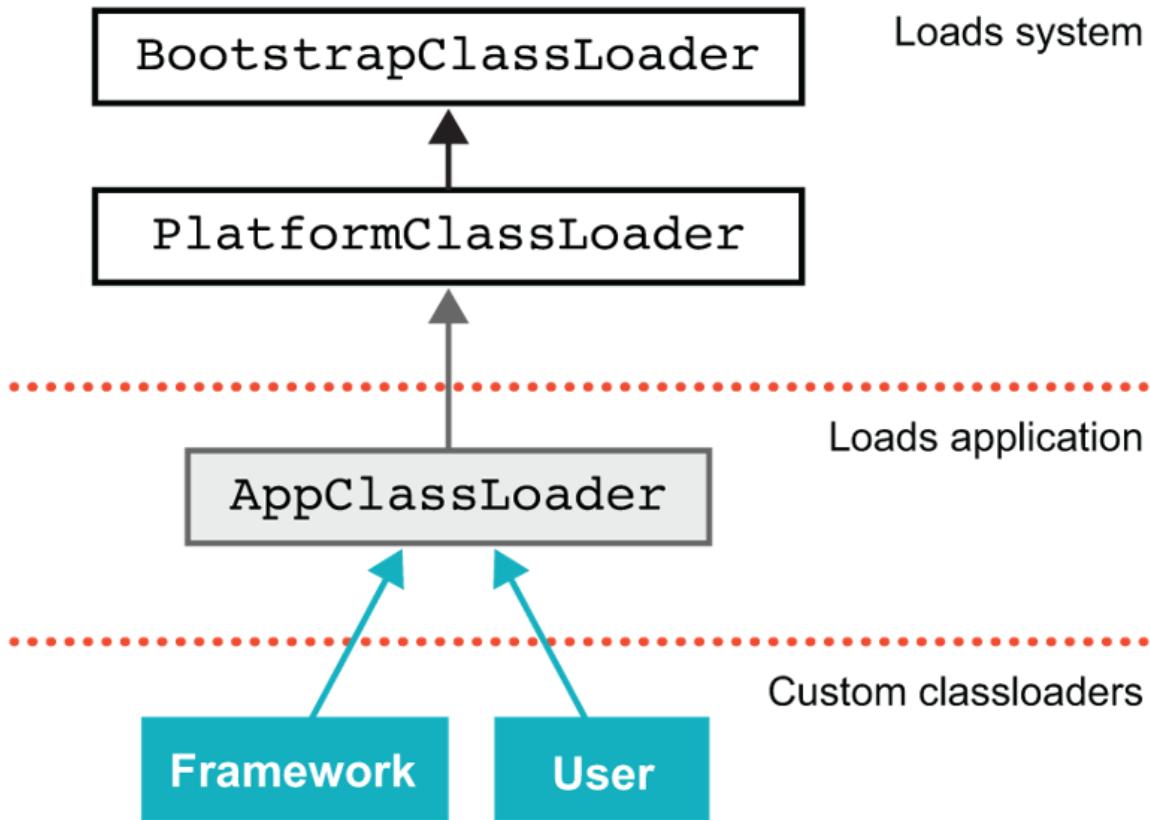


Figure 4.3 Classloader hierarchy

The key to the custom mechanism are the methods `loadClass()` and `findClass()` that are defined on `ClassLoader`. The main entry point is `loadClass()` and a simplified form of the relevant code in `ClassLoader`` is:

```

protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            // ...
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                // ...
                c = findClass(name);
            }
            // ...
        }
        // ...
    }
    return c;
}
}

```

Essentially, this just means that the `loadClass()` mechanism looks to see if the class is already loaded, then asks its parent classloader. If that classloading fails (note the try-catch surrounding the call to `parent.loadClass(name, false)`) then the loading process delegates to `findClass()`. The definition of `findClass()` in `java.lang.ClassLoader` is very simple - it just throws a `ClassNotFoundException`.

At this point, let's return to a question that we posed at the start of the chapter, and explore some of the exception and error types that can be encountered during classloading.

CLASSLOADING EXCEPTIONS

Firstly, the meaning of `ClassNotFoundException` is relatively simple - that the classloader attempted to load the specified class, but was unable to do so. That is, the class was unknown to the JVM at the point where loading was requested - and the JVM was unable to find it.

Next up is `NoClassDefFoundError` - note that this is an *error* rather than an exception. This error indicates that the JVM did know of the existence of the requested class but did not find a definition for it in its internal metadata.

Let's take a quick look at an example:

```

public class ExampleNoClassDef {

    public static class BadInit {
        private static int thisIsFine = 1 / 0;
    }

    public static void main(String[] args) {
        try {
            var init = new BadInit();
        } catch (Throwable t) {
            System.out.println(t);
        }
        var init2 = new BadInit();
        System.out.println(init2.thisIsFine);
    }
}

```

When this runs, we get some output like this:

```

$ java ExampleNoClassDef
java.lang.ExceptionInInitializerError
Exception in thread "main" java.lang.NoClassDefFoundError: Could
    not initialize class ExampleNoClassDef$BadInit
        at ExampleNoClassDef.main(ExampleNoClassDef.java:13)

```

This shows that the JVM tried to load the `BadInit` class but failed to do so. Nevertheless, the program caught the exception and tried to carry on. When the class was encountered for the second time, however, the JVM's internal metadata table showed that the class had been seen - but that a valid class was not loaded.

The JVM effectively implements *negative caching* on a failed classloading attempt - so the loading is not retried, and instead an error (`NoClassDefFoundError`) is thrown.

Another common error is `UnsupportedClassVersionError` - which is triggered when a classloading operation tries to load a class file that was compiled by a higher version of the Java source code compiler than the runtime version.

For example, consider a class compiled with Java 11 that we try to run on Java 8:

```

$ java ScratchImpl
Error: A JNI error has occurred please check your installation and try again
Exception in thread "main" java.lang.UnsupportedClassVersionError:
ScratchImpl has been compiled by a more recent version of the Java
Runtime (class file version 55.0), this version of the Java Runtime
only recognizes class file versions up to 52.0
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
at java.net.URLClassLoader.defineClass(URLClassLoader.java:468)
at java.net.URLClassLoader.access$100(URLClassLoader.java:74)
at java.net.URLClassLoader$1.run(URLClassLoader.java:369)
at java.net.URLClassLoader$1.run(URLClassLoader.java:363)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(URLClassLoader.java:362)
at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
at sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:495)

```

The Java 11 format bytecode may have features in it that are not supported by the runtime, so it is not safe to continue to try to load it. Note that because this is a Java 8 runtime, it does not have modular entries in the stack trace.

Finally, we should also mention `LinkageError` - which is the base class of a hierarchy containing `NoClassDefFoundError`, `VerifyError` and `UnsatisfiedLinkError` as well as several other possibilities.

A FIRST CUSTOM CLASSLOADER

The simplest form of custom classloading is simply to subclass `ClassLoader` and override `findClass()`. This allows us to reuse the `loadClass()` logic that we discussed earlier on and to reduce the complexity in our classloader.

Our first example is the `SadClassLoader` - it doesn't actually do anything but it makes sure that you know that it was technically involved in the process and it wishes you well.

```
public class LoadSomeClasses {

    public static class SadClassLoader extends ClassLoader {
        public SadClassLoader() {
            super(SadClassLoader.class.getClassLoader());
        }

        public Class<?> findClass(String name) throws
            ClassNotFoundException {
            System.out.println("I am very concerned that I
                couldn't find the class");
            throw new ClassNotFoundException(name);
        }
    }

    public static void main(String[] args) {
        if (args.length > 0) {
            var loader = new SadClassLoader();
            for (var name : args) {
                System.out.println(name + " :::");
                try {
                    var clazz = loader.loadClass(name);
                    System.out.println(clazz);
                } catch (ClassNotFoundException x) {
                    x.printStackTrace();
                }
            }
        }
    }
}
```

In our example we set up a very simple classloader and some code that uses it to try to load classes that may or may not already be loaded.

NOTE	One common convention for custom classloaders is to provide a void constructor that calls the superclass constructor and provides the loading classloader as an argument (to become the parent).
-------------	---

Many custom classloaders are not that much more complex than our example - they just override `findClass()` to provide the specific capability that is needed. This could include, for example, looking for the class over the network. In one memorable case, a custom classloader loaded classes by connecting to a database via JDBC and accessing an encrypted binary column to get the bytes that would be used. This was to satisfy an encryption-at-rest requirement for very sensitive code in a highly regulated environment.

It is possible to do more than just override `findClass()`, however. For example, `loadClass()` is not final and so can be overridden, and in fact some custom classloaders do override it precisely to change the general logic we met earlier.

Finally, there is also the method `defineClass()` that is defined on `ClassLoader`.

This method is key to classloading - as it is the user-accessible method that performs the "Loading and Linking" process that we described earlier in the chapter. It takes an array of bytes and turns them into a class object. This is the primary mechanism that is used to load new classes at runtime that are not present on the classpath.

The call to `defineClass()` will only work if it is passed a buffer of bytes that are in the correct JVM class file format - if not then it will fail to load - as either the loading or verification step will fail.

NOTE

This method can be used for advanced techniques such as loading classes that are generated at runtime and have no source code representation. This technique is how the lambda expressions mechanism works in Java - and we will have more to say about this subject in Chapter 16.

The `defineClass()` method is both protected and final, and is defined on `java.lang.ClassLoader` so it can only be accessed by subclasses of `ClassLoader`. Custom classloaders therefore always have access to the basic functionality of `defineClass()` but cannot tamper with the verification or other low-level classloading logic. This last point is important - not being able to change the verification algorithm is a very useful safety feature as it means a poorly-written custom classloader cannot compromise the basic platform security the JVM provides.

In the case of the HotSpot virtual machine (which is by far the most common Java VM implementation), `defineClass()` delegates to the native method `defineClass1()`, which does some basic checks and then calls a C function called `JVM_DefineClassWithSource()`.

This function is an entry point into the JVM - and it provides access into the C++ code of HotSpot. HotSpot uses the C++ `SystemDictionary` to load the new class via the C++ method `ClassFileParser::parseClassFile()`. This is the code that actually runs much of the linking

process - and in particular the verification algorithm.

Once classloading has completed, the bytecode of the methods is placed into Hotspot's metadata objects that represent the methods (they are called *methodOoops*). They are then available for the bytecode interpreter to use. This can be thought of as a method cache conceptually, although the bytecode is actually held inline in the *methodOoops* for performance reasons.

We have already met the `SadClassLoader`, but let's look at another couple of examples of custom classloaders, starting with a look at how classloading can be used to implement *dependency injection* (DI).

EXAMPLE: A DEPENDENCY INJECTION FRAMEWORK

There are two primary concepts that are core to the idea of DI:

- Units of functionality within a system have dependencies and configuration information upon which they rely for proper functioning.
- Many object systems have dependencies that are difficult or clumsy to express.

The picture that should be in your head is of classes that contain behavior, and configuration and dependencies that are external to the objects. This latter part is what is usually referred to as the *runtime wiring* of the objects.

In this example, we'll discuss how a hypothetical DI framework could make use of classloaders to implement runtime wiring. The approach we'll take is like a simplified version of the original implementation of the Spring framework.

However, modern DI frameworks frequently uses another approach which has more compile-time safety - but which has significantly higher complexity and cognitive load to understand. Our example is really for demonstration purposes only.

Let's start by looking at how we'd start an application under our imaginary DI framework:

```
java -cp <CLASSPATH> org.wgjd.DIMain /path/to/config.xml
```

The CLASSPATH must contain the JAR files for the DI framework, and for any classes that are referred to in the config.xml file (along with any dependencies that they have).

For this to be managed under DI, you'll need a config file too, like this:

```
<beans>

<bean id="dao" class="wgjd.ch03.PaymentsDAO">
    <constructor-arg index="0" value="jdbc:postgresql:
        //db.wgjd.org/payments"/>
    <constructor-arg index="1" value="org.postgresql.Driver"/>
</bean>

<bean id="service" class="wgjd.ch03.PaymentService">
    <constructor-arg index="0" ref="dao"/>
</bean>

</beans>
```

In this technique, the DI framework uses the config file to determine which objects to construct. This example needs to make the `dao` and `service` beans, and the framework will call the constructors for each bean.

This means that classloading occurs in two separate phases. The first phase (which is handled by the application classloader) loads the class `DIMain` and any classes that it refers to. Then `DIMain` starts to run and receives the location of the config file as a parameter to `main()`.

At this point, the framework is up and running in the JVM, but the user classes specified in `config.xml` haven't yet been touched. In fact, until `DIMain` examines the config file, the framework has no way of knowing what the classes to be loaded are.

NOTE

This example is hypothetical and illustrative - it would be entirely possible to build a simple DI framework that worked in exactly the manner described here. However the actual implementation of real DI systems is typically more complicated in practice.

To bring up the application configuration specified in `config.xml`, a second phase of classloading is required. This uses a custom classloader.

First, the `config.xml` file is checked for consistency and to make sure it's error-free. Then, if all is well, the custom classloader tries to load the types from the `CLASSPATH`. If any of these fail, the whole process is aborted, causing a runtime error.

If this succeeds, the DI framework can proceed to instantiate the required objects in the correct order and call any setter methods on the created instances. Finally, if all of this completes OK, the application context is up and can begin to run.

EXAMPLE: AN INSTRUMENTING CLASSLOADER

Consider a classloader that alters the bytecode of classes as they're loaded to add extra instrumentation information. When test cases are run against the transformed code, the instrumentation code records which methods and code branches are actually tested by the test cases. From this, the developer can see how thorough the unit tests for a class are.

This approach was the basis of the EMMA testing coverage tool, which is still available from emma.sourceforge.net/ although it is now rather outdated and has not been kept up to date for modern Java versions.

Despite this, it's quite common to encounter frameworks and other code that makes use of specialized classloaders that transform the bytecode as it's being loaded.

NOTE

The technique of modifying bytecode as it is loaded is also seen in the java agent approach, which is used for performance monitoring, observability and other goals - by tools such as New Relic.

We've briefly touched on a couple of use cases for custom classloading. Many other areas of the Java technology space are big users of classloaders and related techniques. These are some of the best-known examples:

- Plugin architectures
- Frameworks (whether vendor or homegrown)
- Class file retrieval from unusual locations (not file systems or URLs)
- Java EE
- Any circumstance where new, unknown code may need to be added after the JVM process has already started running

Let's move on to discuss how the modules system affects classloading and modifies the classic picture that we've just explained.

4.2.2 Modules and classloading

The modules system is designed to operate at a different level to classloading, which is a relatively low-level mechanism within the platform. Modules are about large-scale dependencies between program units and classloading is about the small scale. However, it is important to understand how the two mechanisms intersect and the changes to program startup that have been caused by the arrival of modules.

Recall that when running on a modular JVM then to execute a program, the runtime will compute a module graph and try to satisfy it as a first step. This is referred to as *module resolution* and it derives the transitive closure of the root module and its dependencies.

During this process, additional checks are performed (e.g. no modules with duplicate names, no split packages, etc). The existence of the module graph means that fewer runtime classloading problems are expected - because missing jars on the module path can now be detected before the process even starts fully.

Beyond this, the modules system does not alter classloading much in most cases. There are some advanced possibilities (such as dynamically loading modular implementations of service provider interfaces by using reflection) but those are not likely to be encountered by most developers very often.

4.3 Reflection

One of the key techniques that a well-grounded Java developer should have at their command is *reflection*. This is an extremely powerful capability but many developers struggle with it at first - because it seems alien to the way that most Java developers think about code.

Reflection is the ability to query or *introspect* objects and discover (and use) their capabilities at runtime. It can be thought of as several different things, depending on context:

- A programming language API
- A programming style or technique
- A runtime mechanism that enables the technique
- A property of the language type system

Reflection in an object-oriented system is essentially the idea that the programming environment can represent the types and methods of the program as objects. This is only possible in languages that have a runtime that supports this - and it is a fundamentally dynamic aspect of a language.

When using the reflective style of programming, it is possible to manipulate objects without using their static types at all. This seems like a step backwards, but if we can work with objects without needing to know their static types, then it means that we can build libraries, frameworks and tools that can work with *any* type - including types that did not even exist when our handling code was written.

When Java was a young language, reflection was one of the key technological innovations that it brought to the mainstream. Although other languages (notably Smalltalk) had introduced it much earlier, it was not a common part of many languages at the time Java was released.

4.3.1 Introducing reflection

The abstract description of reflection can often seem confusing or hard to grasp. Let's look at some simple examples in JShell to try to get a more concrete view of what reflection is:

```
jshell> Object o = new Object();
o ==> java.lang.Object@a67c67e

jshell> Class<?> cls = o.getClass();
cls ==> class java.lang.Object
```

This is our first glimpse of reflection - a class object for the type `Object`. In fact, the actual type of `cls` is `Class<Object>` but when we obtain a class object from classloading or `getClass()` we have to handle it using the unknown type, `?`, in the generics:

```
jshell> Class<Object> cls = Object.class;
cls ==> class java.lang.Object

jshell> Class<Object> cls = o.getClass();
|   Error:
| incompatible types: java.lang.Class<capture#1 of ? extends
|   java.lang.Object> cannot be converted to java.lang.Class<java.lang.Object>
|   Class<Object> cls = o.getClass();
|           ^-----^
```

This is because reflection is a dynamic, runtime mechanism, and the true type `Class<Object>` is not known to the source code compiler. This introduces irreducible extra complexity to working with reflection - as we cannot rely on the Java type system to help us very much.

On the other hand, this dynamic nature is the key point of reflection - if we don't know what type something is at compile time and have to treat it in a very general way then we can exploit this flexibility to build an open, extensible system.

NOTE

Reflection produces a fundamentally open system and as we saw in Chapter 2, this can come into conflict with the more encapsulated systems that Java modules try to bring to the platform.

Many familiar frameworks and developer tools rely heavily on reflection to achieve their capabilities, such as debuggers and code browsers. Plugin architectures and interactive environments and REPLs also use reflection extensively. In fact, JShell itself could not be built in a language without a reflection subsystem.

```
jshell> class Pet {
...>     public void feed() {
...>         System.out.println("Feed the pet");
...>     }
...> }
| created class Pet

jshell> var cls = Pet.class;
cls ==> class Pet
```

Now we have an object that represents the class type of `Pet`, we can use to do other actions, such as creating a new instance:

```
jshell> Object o = clz.newInstance();
o ==> Pet@66480dd7
```

The problem we have is that `newInstance()` returns `Object` - which isn't a very useful type. We could of course cast `o` back to `Pet` but this requires us to know ahead of time what types we're working with - which rather defeats the point of the dynamic nature of reflection. So let's try something else:

```
jshell> import java.lang.reflect.Method;
jshell> Method m = clz.getMethod("feed", new Class[0]);
m ==> public void Pet.feed()
```

Now we have an object that represents the method `feed()` - but it represents it as abstract metadata - it is not attached to any specific instance.

The natural thing to do with an object that represents a method is to call it. The class `java.lang.reflect.Method` defines a method `invoke()` that has the effect of calling the method that the `Method` object represents.

NOTE

When working in JShell we're avoiding a lot of exception handling code. When writing regular Java code that uses reflection you will have to deal with the possible exception types in one way or another.

For this call to succeed, then we must provide the right number and types of arguments. This argument list must include the *receiver object* on which the method is being called reflectively (assuming the method is an instance method). In our simple example, this looks like this:

```
jshell> Object ret = m.invoke(o);
Feed the pet
ret ==> null
```

①

- ① The call returns null as the `feed()` method is actually `void`

As well as the `Method` objects, reflection also provides for objects that represent other fundamental concepts within the Java type system and language - such as fields, annotations and constructors. These classes are found in the `java.lang.reflect` package - and some of them (such as `Constructor`) are generic types.

The reflection subsystem also had to be upgraded to deal with modules. Just as classes and methods can be treated reflectively, so there needs to be a reflective API for working with modules. The key class is, perhaps unsurprisingly, `java.lang.Module` and it can be accessed directly from a `Class` object:

```
var module = String.class.getModule();
var descriptor = module.getDescriptor();
```

The descriptor of a module is of type `ModuleDescriptor` and provides a read-only view of the metadata about a module - basically equivalent to the contents of `module-info.class`.

Dynamic capabilities, such as discovery of modules, are also possible in the new reflective API. This is achieved via interfaces such as `ModuleFinder`, but a detailed description of how to work reflectively with the modules system is outside the scope of this book.

4.3.2 Combining classloading and reflection

Let's look at an example that combines classloading and reflection. We won't need a full classloader that obeys the usual `findClass()` and `loadClass()` protocol. Instead, we'll just subclass `ClassLoader` to gain access to the protected `defineClass()` method.

The main method takes a list of filenames, and if they're a Java class, then uses reflection to access each method in turn and detect if it's a native method or not:

```
public class NativeMethodChecker {

    public static class EasyLoader extends ClassLoader {
        public EasyLoader() {
            super(EasyLoader.class.getClassLoader());
        }

        public Class<?> loadFromDisk(String fName) throws IOException {
            var b = Files.readAllBytes(Path.of(fName));
            return defineClass(null, b, 0, b.length);
        }
    }

    public static void main(String[] args) {
        if (args.length > 0) {
            var loader = new EasyLoader();
            for (var file : args) {
                System.out.println(file + " :::");
                try {
                    var clazz = loader.loadFromDisk(file);
                    for (var m : clazz.getMethods()) {
                        if (Modifier.isNative(m.getModifiers())) {
                            System.out.println(m.getName());
                        }
                    }
                } catch (IOException | ClassFormatError x) {
                    System.out.println("Not a class file");
                }
            }
        }
    }
}
```

These types of examples can be fun to explore the dynamic nature of the Java platform and to learn how the Reflection API works. However, it's important that a well-grounded Java developer is also conscious of the limitations and occasional frustrations that can occur when working reflectively.

4.3.3 Problems with reflection

The Reflection API has been part of the Java platform since 1.1 (1996) and in the almost 25 years since its arrival a number of issues and weaknesses have come to light. Some of these inconveniences include:

- It's a very old API - with array types everywhere (it predates the Java Collections)
- Figuring out which method overload to call is painful
- API has two different methods `getMethod()` and `getDeclaredMethod()` to access methods reflectively
- API provides the `setAccessible()` method which can be used to ignore access control
- Exception handling is complex for reflective calls - checked exceptions get elevated to runtime exceptions
- Boxing and unboxing is necessary to make reflective calls that pass or return primitives
- Primitive types require placeholder class objects, e.g. `int.class` - which is actually of type `Class<Integer>`
- `void` methods require the introduction of the `java.lang.Void` type

As well as the various awkward corners in the API, Java Reflection has always suffered from poor performance - for several reasons, including unfriendliness to the JVM's JIT compiler.

NOTE

Solving the problem of reflective call performance was one of the major reasons for the addition of the Method Handles API, which we will meet in Chapter 15.

There is one final problem with reflection, which is perhaps more of a philosophical problem (or anti-pattern): Developers frequently encounter reflection as one of the first truly advanced techniques that they meet when levelling up in Java.

As a result it can become overused or a *Golden Hammer* technique - used to implement systems which are excessively flexible or which display an internal mini-framework which is not really needed (sometimes called the *Inner Framework* anti-pattern). Such systems are often very configurable but at the expense of encoding the domain model into configuration rather than directly in the domain types.

Reflection is a great technique and one that the well-grounded Java developer should have in their toolbox, but it is not suitable for every situation and most developers will probably only ever need to use it sparingly.

This concludes our discussion of classloading and reflection, so let's move on to discuss class file and bytecode in more detail.

4.4 Examining class files

Class files are binary blobs, so they aren't easy to work with directly. But there are many circumstances in which you'll find that investigating a class file is necessary.

Imagine that your application needs additional methods to be made public to allow better runtime monitoring (such as via JMX). The recompile and redeploy seems to complete fine, but when the management API is checked, the methods aren't there. Additional rebuild and redeploy steps have no effect.

To debug the deployment issue, you may need to check that javac has produced the class file that you think it has. Or you may need to investigate a class that you don't have source for and where you suspect the documentation is incorrect.

For these and similar tasks, you must make use of tools to examine the contents of class files. Fortunately, the standard Oracle JVM ships with a tool called `javap`, very handy for peeking inside and disassembling class files.

We'll start off by introducing `javap` and some of the basic switches it provides to examine aspects of class files. Then we'll discuss some of the representations for method names and types that the JVM uses internally. We'll move on to take a look at the constant pool—the JVMs "box of useful things"—which plays an important role in understanding how bytecode works.

4.4.1 Introducing `javap`

`javap` can be used for numerous useful tasks, from seeing what methods a class declares to printing the bytecode. Let's examine the simplest form of `javap` usage, as applied to the classloading example from earlier in the chapter:

```
$ javap LoadSomeClasses.class
Compiled from "LoadSomeClasses.java"
public class LoadSomeClasses {
    public LoadSomeClasses();
    public static void main(java.lang.String[]);
}
```

The inner class has been compiled out into a separate class, so we need to also look at that one:

```
$ javap LoadSomeClasses\$\$SadClassLoader.class
Compiled from "LoadSomeClasses.java"
public class LoadSomeClasses\$\$SadClassLoader extends java.lang.ClassLoader {
    public LoadSomeClasses\$\$SadClassLoader();
    public java.lang.Class<?> findClass(java.lang.String) throws
        java.lang.ClassNotFoundException;
}
```

By default, `javap` shows the public, protected, and default access (package-protected) visibility methods. The `-p` switch also shows the private methods and fields.

4.4.2 Internal form for method signatures

The JVM uses a slightly different form for method signatures internally than the human-readable form displayed by `javap`. As we delve deeper into the JVM, you'll see these internal names more frequently. If you're keen to keep going, you can jump ahead, but remember that this section's here—you may need to refer to it from later sections and chapters.

In the compact form, type names are compressed. For example, `int` is represented by `I`. These compact forms are sometimes referred to as *type descriptors*. A complete list is provided in table 4.1.

Table 4.1 Type descriptors

Descriptor	Type
B	byte
C	char (a 16-bit Unicode character)
D	double
F	float
I	int
J	long
L<type name>;	Reference type (such as Ljava/lang/String; for a string)
S	short
Z	boolean
[array-of

In some cases, the type descriptor can be longer than the type name that appears in source code (for example, `Ljava/lang/Object;` is longer than `Object`, but the type descriptors are fully qualified so they can be directly resolved).

`javap` provides a helpful switch, `-s`, which will output the type descriptors of signatures for you, so you don't have to work them out using the table. You can use a slightly more advanced invocation of `javap` to show the signatures for some of the methods we looked at earlier:

```
$ javap -s LoadSomeClasses.class
Compiled from "LoadSomeClasses.java"
public class LoadSomeClasses {
    public LoadSomeClasses();
        descriptor: ()V

    public static void main(java.lang.String[]);
        descriptor: ([Ljava/lang/String;)V
}
```

and for the inner class:

```
$ javap -s LoadSomeClasses$SadClassLoader.class
Compiled from "LoadSomeClasses.java"
public class LoadSomeClasses$SadClassLoader extends java.lang.ClassLoader {
    public LoadSomeClasses$SadClassLoader();
        descriptor: ()V

    public java.lang.Class<?> findClass(java.lang.String) throws
        java.lang.ClassNotFoundException;
        descriptor: (Ljava/lang/String;)Ljava/lang/Class;
}
```

As you can see, each type in a method signature is represented by a type descriptor.

In the next section, you'll see another use of type descriptors. This is in a very important part of the class file—the constant pool.

4.4.3 The constant pool

The constant pool is an area that provides handy shortcuts to other (constant) elements of the class file. If you've studied languages like C or Perl, which make explicit use of symbol tables, you can think of the constant pool as being a somewhat-similar JVM concept.

Let's use a very simple example to demonstrate the constant pool, so we don't swamp ourselves with detail. The next listing shows a simple "playpen" or "scratchpad" class. This provides a way to quickly test out a Java syntax feature or library, by writing a small amount of code in run().

Listing 4.1 Sample playpen class

```
package wgjd.ch04;

public class ScratchImpl {

    private static ScratchImpl inst = null;

    private ScratchImpl() {

    }

    private void run() {

    }

    public static void main(String[] args) {
        inst = new ScratchImpl();
        inst.run();
    }
}
```

To see the information in the constant pool, you can use `javap -v`. This prints a lot of additional information — much more than just the constant pool — but let's focus on the constant pool entries for the playpen.

```

#1 = Class #2 // wgjd/ch04/ScratchImpl

#2 = Utf8 wgjd/ch04/ScratchImpl

#3 = Class #4 // java/lang/Object

#4 = Utf8 java/lang/Object

#5 = Utf8 inst

#6 = Utf8 Lwgjd/ch04/ScratchImpl;

#7 = Utf8 <clinit>

#8 = Utf8 ()V

#9 = Utf8 Code

#10 = Fieldref #1.#11 // wgjd/ch04/ScratchImpl.inst:Lwgjd/ch04/ScratchImpl;

#11 = NameAndType #5:#6 // instance:Lwgjd/ch04/ScratchImpl;

#12 = Utf8 LineNumberTable

#13 = Utf8 LocalVariableTable

#14 = Utf8 <init>

#15 = Methodref #3.#16 // java/lang/Object."<init>":()V

#16 = NameAndType #14:#8 // "<init>":()V

#17 = Utf8 this

#18 = Utf8 run

#19 = Utf8 ([Ljava/lang/String;)V

#20 = Methodref #1.#21 // wgjd/ch04/ScratchImpl.run:()V

#21 = NameAndType #18:#8 // run:()V

#22 = Utf8 args

#23 = Utf8 [Ljava/lang/String;

#24 = Utf8 main

#25 = Methodref #1.#16 // wgjd/ch04/ScratchImpl."<init>":()V

#26 = Methodref #1.#27 // wgjd/ch04/ScratchImpl.run:([Ljava/lang/String;)V

#27 = NameAndType #18:#19 // run:([Ljava/lang/String;)V

#28 = Utf8 SourceFile

#29 = Utf8 ScratchImpl.java

```

As you can see, constant pool entries are typed. They also refer to each other, so, for example, an entry of type `Class` will refer to an entry of type `Utf8`. A `Utf8` entry means a string, so the `Utf8` entry that a `Class` entry points out will be the name of the class.

Table 4.2 shows the set of possibilities for entries in the constant pool. Entries from the constant pool are sometimes discussed with a `CONSTANT_` prefix, such as `CONSTANT_Class`. This is to

make it clear that they are not Java types, in situations where they could be confused.

Table 4.2 Constant pool entries

Name	Description
Class	A class constant. Points at the name of the class (as a <code>Utf8</code> entry).
Fieldref	Defines a field. Points at the Class and <code>NameAndType</code> of this field.
Methodref	Defines a method. Points at the Class and <code>NameAndType</code> of this field.
InterfaceMethodref	Defines an interface method. Points at the Class and <code>NameAndType</code> of this field.
String	A string constant. Points at the <code>Utf8</code> entry that holds the characters.
Integer	An integer constant (4 bytes).
Float	A floating-point constant (4 bytes).
Long	A long constant (8 bytes).
Double	A double-precision floating-point constant (8 bytes).
NameAndType	Describes a name and type pair. The type points at the <code>Utf8</code> that holds the type descriptor for the type.
Utf8	A stream of bytes representing <code>Utf8</code> -encoded characters.
InvokeDynamic	Part of invokedynamic mechanism - see Chapter 16
MethodHandle	Part of invokedynamic mechanism - see Chapter 16
MethodType	Part of invokedynamic mechanism - see Chapter 16

Using this table, you can look at an example constant resolution from the constant pool of the playpen. Consider the `Fieldref` at entry #10.

To resolve a field, you need a name, a type, and a class where it resides: #10 has the value #1.#11, which means constant #11 from class #1. It's easy to check that #1 is indeed a constant of type `class`, and #11 is a `NameAndType`. #1 refers to the `ScratchImpl` Java class itself, and #11 is #5:#6—a variable called `inst` of type `ScratchImpl`. So, overall, #10 refers to the static variable `inst` in the `ScratchImpl` class itself (which you might have been able to guess from the output in listing 3.6).

In the verification step of classloading, there's a step to check that the static information in the class file is consistent. The preceding example shows the kind of integrity check that the runtime will perform when loading a new class.

We've discussed some of the basic anatomy of a class file. Let's move on to the next topic, where we'll delve into the world of bytecode. Understanding how source code is turned into bytecode will help you gain a better understanding of how your code will run. In turn, this will lead to more insights into the platform's capabilities when we reach chapter 6 and beyond.

4.5 Bytecode

Bytecode has been a somewhat behind-the-scenes player in our discussion so far. Let's start by reviewing what we've already learned about it:

- Bytecode is an intermediate representation of a program—halfway between human readable source and machine code.
- Bytecode is produced by javac from Java source code files.
- Some high-level language features have been compiled away and don't appear in bytecode. For example, Java's looping constructs (for, while, and the like) are gone, turned into bytecode branch instructions.
- Each opcode is represented by a single byte (hence the name *bytecode*).
- Bytecode is an abstract representation, not "machine code for an imaginary CPU."
- Bytecode can be further compiled to machine code, usually "just in time."

When explaining bytecode, there can be a slight chicken-and-egg problem. In order to fully understand what's going on, you need to understand both bytecode and the runtime environment that it executes in.

This is a rather circular dependency, so to solve it, we'll start by diving in and looking at a relatively simple example. Even if you don't get everything that's in this example on the first pass, you can come back to it after you've read more about bytecode in the following sections.

After the example, we'll provide some context about the runtime environment, and then catalogue the JVM's opcodes, including bytecodes for arithmetic, invocation, shortcut forms, and more. At the end, we'll round off with another example, based on string concatenation. Let's get started by looking at how you can examine bytecode from a .class file.

4.5.1 Disassembling a class

Using javap with the -c switch, you can disassemble classes. In our example, we'll use the scratchpad class we met in listing 3.5. The main focus will be to examine the bytecode contained within methods. We'll also use the -p switch so we can see bytecode that's contained within private methods.

Let's work section by section — there's a lot of information in each part of javap's output, and it's easy to become overwhelmed. First, the header. There's nothing terribly unexpected or exciting in here:

```
$ javap -c -p wgjd/ch04/ScratchImpl.class
Compiled from "ScratchImpl.java"
public class wgjd.ch04.ScratchImpl extends java.lang.Object {
    private static wgjd.ch04.ScratchImpl inst;
```

Next is the static block. This is where variable initialization is placed, so this represents initializing `inst` to null. The keen-eyed reader might guess that `putstatic` could be a bytecode that puts a value in a static field.

```
static {};

Code:
 0: aconst_null
 1: putstatic #10 // Field inst:Lwgjd/ch04/ScratchImpl;
 4: return
```

The numbers in the preceding code represent the offset into the bytecode stream since the start of the method. So byte 1 is the `putstatic` opcode, and bytes 2 and 3 represent a 16-bit index into the constant pool. In this case, the 16-bit index is the value 10, which means that the value (in this case null) will be stored in the field indicated by constant pool entry #10. Byte 4 from the start of the bytecode stream is the `return` opcode—the end of the block of code.

Next up is the constructor.

```
private wgjd.ch04.ScratchImpl();

Code:
 0: aload_0
 1: invokespecial #15 // Method java/lang/Object."<init>":()V
 4: return
```

Remember that in Java, the void constructor will always implicitly call the superclass constructor. Here you can see this in the bytecode — it's the `invokespecial` instruction. In general, any method call will be turned into one of the VM's 5 different `invoke` instructions.

There's basically no code in the `run()` method, as this is just an empty scratchpad class. This method immediately returns to caller and does not pass a value back (which is correct, because the method returns `void`):

```
private void run();

Code:
 0: return
```

In the main method, we initialize `inst` and do a bit of object creation. This demonstrates some very common basic bytecode patterns that we can learn to recognize:

```
public static void main(java.lang.String[]);

Code:
 0: new #1 // class wgjd/ch04/ScratchImpl
 3: dup
 4: invokespecial #21 // Method "<init>":()V
```

This pattern of three bytecode instructions — `new`, `dup`, and `invokespecial` of a method called `<init>` — always represents the creation of a new instance.

The new opcode only allocates memory for a new instance. The `dup` opcode duplicates the element that's on top of the stack. To finish fully creating the object, we need to call the body of the constructor. The `<init>` method contains the code for the constructor body, so we call that code block with `invokespecial`. Let's look at the remaining bytecodes for the main method:

```
7: putstatic #10 // Field inst:Lwgjd/ch04/ScratchImpl;
10: getstatic #10 // Field inst:Lwgjd/ch04/ScratchImpl;
13: invokevirtual #22 // Method run:()V
16: return
```

Instruction 7 saves the singleton instance that has been created. Instruction 10 puts it back on top of the stack, so that instruction 13 can call a method on it. This is done with the `invokevirtual` opcode, which carries out Java's "standard" dispatch for instance methods.

NOTE

In general, the bytecode produced by javac is quite a simple representation—it isn't highly optimized. The overall strategy is that JIT compilers do a lot of optimizing, so it helps if they have a relatively plain and simple starting point. The expression, "Bytecode should be dumb" describes the general feeling of VM implementers toward the bytecode produced from source languages.

The `invokevirtual` opcode includes checking for overrides of the method in the object's inheritance hierarchy. You might notice that this is a bit odd, as private methods can't be overridden. You might guess that the source code compiler could actually emit `invokespecial` instead of `invokevirtual` for private methods. In fact, this used to be the case and was only changed in recent versions of Java - for the details see the section on Nestmates in Chapter 16.

Let's move on to discuss the runtime environment that bytecode needs. After that, we'll introduce the tables that we'll use to describe the major families of bytecode instructions—load/store, arithmetic, execution control, method invocation, and platform operations. Then we'll discuss possible shortcut forms of opcodes, before moving on to another example.

4.5.2 The runtime environment

Understanding the operation of the stack machine that the JVM uses is critical to understanding bytecode.

One of the most obvious ways that the JVM doesn't look like a hardware CPU (such as an x64 or ARM chip) is that the JVM doesn't have processor registers, and instead uses a stack for all calculations and operations.

This is referred to as the *evaluation stack* (it's officially called the *operand stack* in the VM Specification, and we'll use the two terms interchangeably).

Figure 4.4 shows how the evaluation stack might be used to perform an addition operation on two int constants. We're showing the equivalent JVM bytecode on the right hand side - we'll meet this bytecode later in the chapter, so don't worry if it doesn't make complete sense right now.

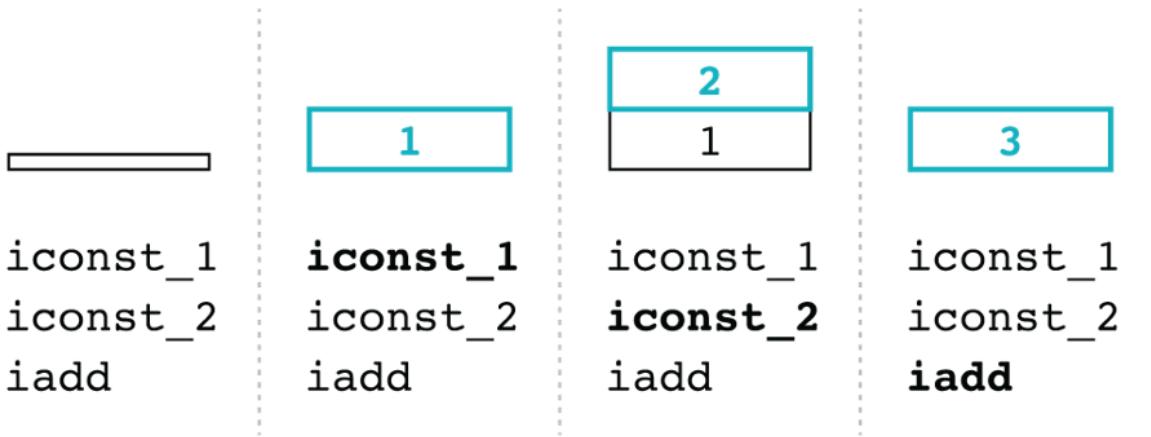


Figure 4.4 Using a stack for numerical calculations

As we discussed earlier in this chapter, when a class is linked into the running environment, its bytecode will be checked, and a lot of that verification boils down to analyzing the pattern of types on the stack.

NOTE

Manipulations of the values on the stack will only work if the values on the stack have the correct types. Undefined or bad things could happen if, for example, we pushed a reference to an object onto the stack and then tried to treat it as an int and do arithmetic on it.

The verification phase of classloading performs extensive checks to ensure that methods in newly loaded classes don't try to abuse the stack. This prevents a malformed (or deliberately evil) class from ever being accepted by the system and causing problems.

As a method runs, it needs an area of memory to use as an evaluation stack, for computing new values. In addition, every running thread needs a call stack that records which methods are currently in flight (the stack that would be reported by a stack trace). These two stacks will interact in some cases. Consider this bit of code:

```
return 3 + petRecords.getNumberOfPets("Ben");
```

To evaluate this, you put 3 on the operand stack. Then you need to call a method to calculate how many pets Ben has. To do this, you push the receiver object (the one you're calling the method on — `petRecords()` in this example) onto the evaluation stack, followed by any arguments you want to pass in.

Then the `getNumberOfPets()` method will be called using an `invoke` opcode, which will cause control to transfer to the called method, and the just-entered method to appear in the call stack. But as you enter the new method, you need to start using a different operand stack, so that the values already on the caller's operand stack can't possibly affect results calculated in the called method.

As `getNumberOfPets()` completes, the return value is placed onto the operand stack of the caller, as part of the process whereby `getNumberOfPets()` is removed from the call stack. Then the addition operation can take the two values and add them.

Let's now turn to examining bytecode. This is a large subject, with lots of special cases, so we're going to present an overview of the main features rather than a complete treatment.

4.5.3 Introduction to opcodes

JVM bytecode consists of a sequence of operation codes (opcodes), possibly with some arguments following each instruction. Opcodes expect to find the stack in a given state, and transform the stack, so that the arguments are removed and results placed there instead.

Each opcode is denoted by a single-byte value, so there are at most 255 possible opcodes. Currently, only around 200 are used. This is too many for us to list exhaustively, but fortunately most opcodes fit into one of a number of families. We'll discuss each family in turn, to help you get a feel for them. There are a number of operations that don't fit cleanly into any of the families, but they tend to be encountered less often.

NOTE

The JVM isn't a purely object-oriented runtime environment — it has knowledge of primitive types. This shows up in some of the opcode families — some of the basic opcode types (such as `store` and `add`) are required to have a number of variations that differ depending on the primitive type they're acting upon.

The opcode tables have four columns:

- *Name* — This is a general name for the type of opcode. In many cases there will be several related opcodes that do similar things.
- *Args* — The arguments that the opcode takes. Arguments that start with `i` are (unsigned) bytes that are used to form a lookup index e.g. in the constant pool or local variable table.

NOTE

To make longer indices, bytes are joined together, so that `i1, i2` means "make a 16-bit index out of these two bytes" via bit shifting and addition `((i1 << 8) + i2)` - or in the slightly more explicit Rust code: `((i1 as u16) << 8) + i2 as u16`

If an arg is shown in brackets, it means that not all forms of the opcode will use it.

- *Stack layout*—This shows the state of the stack before and after the opcode has executed. Elements in brackets indicate that not all forms of the opcode use them, or that the elements are optional (such as for invocation opcodes).
- *Description*—What the opcode does.

Let's look at an example of a row from table 4.3, by examining the entry for the getfield opcode. This is used to read a value from a field of an object.

<code>getfield</code>	<code>i1, i2</code>	<code>[obj]</code>	<code>[val]</code>	Gets the field at the constant pool index specified from the object on top of the stack.
-----------------------	---------------------	--------------------	--------------------	--

The first column gives the name of the opcode — `getfield`. The next column says that there are two arguments that follow the opcode in the bytecode stream. These arguments are put together to make a 16-bit value that is looked up in the constant pool to see which field is wanted (remember that constant pool indexes are always 16-bit).

The stack layout column shows you that after the index has been looked up in the constant pool of the class of the object on top of the stack, the object is removed and is replaced by the value of that field for the object that was on top of the stack.

This pattern of removing object instances as part of the operation is just a way to make bytecode compact, without lots of tedious cleanup and remembering to remove object instances that you're finished with.

4.5.4 Load and store opcodes

The family of load and store opcodes is concerned with loading values onto the stack, or retrieving them. Table 4.4 shows the main operations in the load/store family.

Table 4.3 Load and store opcodes

Name	Args	Stack layout	Description
load	(i1)	[] [val]	Loads a value (primitive or reference) from a local variable onto the stack. Has shortcut forms and type-specific variants.
ldc	i1	[] [val]	Loads a constant from the pool onto the stack. Has type-specific and wide variants.
store	(i1)	[val] []	Stores a value (primitive or reference) in a local variable, removing it from the stack in the process. Has shortcut forms and type-specific variants.
dup		[val] [val, val]	Duplicates the value on top of the stack. Has variant forms.
getfield	i1, i2	[obj] [val]	Gets the field at the constant pool index specified from the object on top of the stack.
putfield	i1, i2	[obj, val] []	Puts the value into the object's field at the specified constant pool index.
getstatic	i1, i2	[] [val]	Gets the value of the static field at the constant pool index specified.
putstatic	i1, i2	[val] []	Puts the value into the static field at the specified constant pool index.

As we noted earlier, there are a number of different forms of the load and store instructions. For example, there is a `dload` opcode to load a double onto the stack from a local variable, and an `astore` opcode to pop an object reference off the stack and into a local variable.

Let's do a quick example of `getfield` and `putfield`. This simple class:

```
public class Scratch {
    private int i;

    public Scratch() {
        i = 0;
    }

    public int getI() {
        return i;
    }

    public void setI(int i) {
        this.i = i;
    }
}
```

will decompile the getter and setter as:

```
public int getI();
Code:
 0: aload_0
 1: getfield      #7           // Field i:I
 4: ireturn

public void setI(int);
Code:
 0: aload_0
 1: iload_1
 2: putfield      #7           // Field i:I
 5: return
```

which shows how the stack is used to hold temporary variables before transferring them to heap storage.

4.5.5 Arithmetic opcodes

These opcodes perform arithmetic on the stack. They take arguments from the top of the stack and perform the required calculation on them. The arguments (which are always primitive types) must always match exactly, but the platform provides a wealth of opcodes to cast one primitive type to another. Table 4.5 shows the basic arithmetic operations.

Table 4.4 Arithmetic opcodes

Name	Args	Stack layout	Description
add		[val1, val2] [res]	Adds two values (which must be of the same primitive type) from the top of the stack and stores the result on the stack. Has shortcut forms and type-specific variants.
sub		[val1, val2] [res]	Subtracts two values (of the same primitive type) from top of the stack. Has shortcut forms and type-specific variants.
div		[val1, val2] [res]	Divides two values (of the same primitive type) from top of the stack. Has shortcut forms and type-specific variants.
mul		[val1, val2] [res]	Multiplies two values (of the same primitive type) from top of the stack. Has shortcut forms and type-specific variants.
(cast)		[value] [res]	Casts a value from one primitive type to another. Has forms corresponding to each possible cast.

The cast opcodes have very short names, such as `i2d` for an `int` to `double` cast. In particular, the word *cast* doesn't appear in the names, which is why it's in parentheses in the table.

4.5.6 Execution flow control opcodes

As mentioned earlier, the control constructs of high-level languages aren't present in JVM bytecode. Instead, flow control is handled by a small number of primitives, which are shown in table 4.6.

Table 4.5 Execution control opcodes

Name	Args	Stack layout	Description
if	b1, b2	[val1, val2] [] or [val1] []	If the specific condition matches, jump to the specified branch offset.
goto	b1, b2	[] []	Unconditionally jump to the branch offset. Has wide form.
tableswitch	{depends}	[index] []	Used to implement switch.
lookupswitch	{depends}	[key] []	Used to implement switch.

Like the index bytes used to look up constants, the `b1`, `b2` args are used to construct a bytecode location within this method to jump to. They cannot be used to jump outside of the method - this is checked at classloading time and would cause the class to fail verification.

The family of `if` opcodes is a little larger than you might expect - it has over 15 instructions to handle the various source code possibilities (e.g. numeric comparison, reference equality, etc).

NOTE This family also contains two deprecated instructions, `jsr` and `ret` which are no longer produced by `javac` and are illegal in modern Java versions.

There is a wide form of the `goto` instruction (`goto_w`) that takes 4 bytes of arguments, and construct an offset, which can be larger than 64 KB. This isn't often needed - as it would only apply to very, very large methods (and such methods have other problems, such as being too large to be JIT-compiled). There is also `ldc_w` which can be used to address very large constant pools.

4.5.7 Invocation opcodes

The invocation opcodes comprise four opcodes for handling general method calling, plus the unusual `invokedynamic` opcode, which was only added in Java 7, which we'll discuss in more detail in Chapter 16. The five method invocation opcodes are shown in table 4.7.

Table 4.6 Invocation opcodes

Name	Args	Stack layout	Description
invokestatic	i1, i2	[(val1, ...)] []	Calls a static method.
invokevirtual	i1, i2	[obj, (val1, ...)] []	Calls a "normal" instance method.
invokeinterface	i1, i2, count, 0	[obj, (val1, ...)] []	Calls an interface method.
invokespecial	i1, i2	[obj, (val1, ...)] []	Calls a "special" instance method.
invokedynamic	i1, i2, 0, 0	[val1, ...] []	Dynamic invocation; See Chapter 16.

It's easiest to see the difference between these opcodes with an extended example.

```
long time = System.currentTimeMillis();

// This explicit typing is deliberate... read on
HashMap<String, String> hm = new HashMap<>();
hm.put("now", "bar");

Map<String, String> m = hm;
m.put("foo", "baz");
```

Let's use `javap -c` to look at the bytecode for this:

```
Code:
 0: invokestatic #2 // Method java/lang/System.currentTimeMillis:()J
 3: lstore_1
 4: new           #3 // class java/util/HashMap
 7: dup
 8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc           #5 // String now
15: ldc           #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:(
                  //Ljava/lang/Object;Ljava/lang/Object;)
                  //Ljava/lang/Object;
20: pop
21: aload_3
22: astore        4
24: aload        4
26: ldc           #8 // String foo
28: ldc           #9 // String baz
30: invokeinterface #10, 3 // InterfaceMethod java/util/Map.put:(
                           //Ljava/lang/Object;Ljava/lang/Object;)
                           //Ljava/lang/Object;
35: pop
```

As we discussed earlier, the Java method calls are actually turned into one of several possible `invoke*` bytecodes - let's take a closer look

```
0: invokestatic #2 // Method java/lang/System.currentTimeMillis:()J
3: lstore_1
```

The static call to `System.currentTimeMillis()` is turned into an `invokestatic` that appears at position 0 in the bytecode. This method takes no parameters, so nothing needs to be loaded

onto the evaluation stack before the call is dispatched.

Next, the two bytes 00 02 appear in the byte stream. These are combined into a 16-bit number that is used as an offset into the Constant Pool.

The decompiler helpfully includes a comment that lets the user know which method offset #02 corresponds to - in this case, as expected, it's the method `System.currentTimeMillis()`.

On return, the result of the call is placed on the stack, and at offset 3 we see the single, argument-less opcode `lstore_1` that saves this return value off into local variable 1.

Human readers are, of course, able to see that the variable `time` is never used again. However, one of the design goals of javac is to represent the contents of the Java source code as faithfully as possible - whether it makes sense or not. Therefore, the return value of `System.currentTimeMillis()` is stored, even though it is not used after this point in the program.

This is "dumb bytecode" in action - remember that from the point of view of the platform the class file format is the input format to the compiler that really matters - the JIT compiler.

```

4: new           #3 // class java/util/HashMap
7: dup
8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc           #5 // String now
15: ldc           #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:
                  //Ljava/lang/Object;Ljava/lang/Object;
                  //Ljava/lang/Object;
20: pop

```

Bytecodes 4 to 10 create a new `HashMap` instance, before instruction 11 saves a copy of it into a local variable. Next up, instructions 12 to 16 set up the stack with the `HashMap` object and the arguments for the call to `put()`. The actual invocation of the `put()` method is performed by instructions 17 to 19.

The invoke opcode used this time is `invokevirtual`. This is because the static type of the local variable was declared as `HashMap` - a class type. We will see what will happen if the local variable is declared as `Map` in a moment.

An instance method call differs from a static method call because a static call does not have an instance on which the method is called (sometimes called the receiver object).

NOTE

In bytecode, an instance call must be set up by placing the receiver and any call arguments on the evaluation stack and then issuing the invoke instruction.

In this case the return value from `put()` is not used, so instruction 20 discards it.

```

21: aload_3
22:  astore      4
24:  aload      4
26:  ldc         #8 // String foo
28:  ldc         #9 // String baz
30:  invokeinterface #10,  3 //InterfaceMethod java/util/Map.put:(  

                           //Ljava/lang/Object;Ljava/lang/Object;)  

                           //Ljava/lang/Object;
35:  pop

```

The sequence of bytes from 21 to 25 seems rather odd at first glance. The `HashMap` instance that we created at 4, and saved to local variable 3 at instruction 11 is now loaded back onto the stack, and then a copy of the reference is saved to local var 4. This process removes it from the stack, so it must be reloaded (from variable 4) before use.

This shuffling occurs because in the original Java code, we create an additional local variable (of type `Map` rather than `HashMap`), even though it always refers to the same object as the original variable. This is another example of the bytecode staying as close as possible to the original source code.

After the stack and variable shuffling, the values to be placed in the map are loaded at instructions 26 to 29. With the stack prepared with receiver and arguments, the call to `put()` is dispatched at instruction 30. This time, the opcode is `invokeinterface` - even though the exact same method is actually being called. This is because the Java local variable is of type `Map` - an interface type.

Once again, the return value from `put()` is discarded, via the `pop` at instruction 35.

As well as knowing which Java method invocations turn into which operations, there are a couple of other wrinkles to notice about the invocation opcodes. First off is that `invokeinterface` has extra parameters. These are present for historical and backward compatibility reasons and aren't used these days. The two extra zeros on `invokedynamic` are present for forward-compatibility reasons.

The other important point is the distinction between a regular and a special instance method call. A regular call is *virtual*, which means that the exact method to be called is looked up at runtime using the standard Java rules of method overriding.

However, there are a couple of special cases — including calls to a superclass method. In these cases, you don't want the override rules to be triggered, so you need a different invocation opcode to allow for this case.

This is why the opcode set needs an opcode for invocation of methods without the override mechanism - `invokespecial` - which instead indicates exactly which method will be called.

4.5.8 Platform operation opcodes

The platform operation family of opcodes includes the new opcode, for allocating new object instances, and the thread-related opcodes, such as `monitorenter` and `monitorexit`. The details of this family can be seen in table 3.7.

Table 4.7 Platform opcodes

Name	Args	Stack layout	Description
<code>new</code>	<code>i1, i2</code>	<code>[] [obj]</code>	Allocates memory for a new object, of the type specified by the constant at the specified index.
<code>monitorenter</code>		<code>[obj] []</code>	Locks an object. See Chapter 4.
<code>monitorexit</code>		<code>[obj] []</code>	Unlocks an object. See Chapter 4.

The platform opcodes are used to control certain aspects of object lifecycle, such as creating new objects and locking them. It's important to notice that the new opcode only allocates storage. The high-level conception of object construction also includes running the code inside the constructor.

At the bytecode level, the constructor is turned into a method with a special name — `<init>`. This can't be called from user Java code, but can be called by bytecode. This leads to the distinctive bytecode pattern that directly corresponds to object creation — a `new` followed by a `dup` followed by an `invokespecial` to call the `<init>` method - as we saw earlier on.

The `monitorenter` and `monitorexit` bytecodes correspond to the start and end of a synchronized block.

4.5.9 Shortcut opcode forms

Many of the opcodes have shortcut forms to save a few bytes here and there. The general pattern is that certain local variables will be accessed much more frequently than others, so it makes sense to have a special opcode that means "do the general operation directly on the local variable" rather than having to specify the local variable as an argument.

This gives rise to opcodes such as `aload_0` and `dstore_2` within the load/store family which are 1 byte shorter than the equivalent byte sequences `aload 00` or `dstore 02`.

NOTE

One byte saved may not sound like much, but it adds up over the entire class. Java's original use case was applets - which were often downloaded over dialup modems, at speeds of 28.8 kilobits per second. With only that speed of bandwidth, it was important to save bytes wherever possible.

To become a truly well-grounded Java developer, you should run `javap` against some of your own classes and learn to recognize common bytecode patterns. For now, with this brief introduction to bytecode under our belts, let's move on to tackle our next subject.

4.6 Summary

In this chapter, we've taken a quick first look into bytecode and classloading. We've dissected the class file format and taken a brief tour through the runtime environment that the JVM provides. By learning more about the internals of the platform, you'll become a better developer.

These are some of the things that we hope you've learned from this chapter:

- The class file format and classloading are central to the operation of the JVM. They're essential for any language that wants to run on the VM.
- The various phases of classloading enable both security and performance features at runtime.
- Reflection is a major feature and extremely powerful
- JVM bytecode is organized into families with related functionality.
- Using `javap` to disassemble class files can help you understand the low level

It's time to move on to the next big topic that will help you stand out as a well-grounded Java developer. By reading the next chapter, you'll learn the fundamentals of Java concurrency and how threading and the CPU hardware model interact.

Java concurrency fundamentals



This chapter covers

- Concurrency theory
- Block-structured concurrency
- Synchronization
- The Java Memory Model (JMM)
- Concurrency support in bytecode

Java has two, mostly separate concurrency APIs - the older API, which is usually called *block-structured concurrency* or *synchronization-based concurrency* or even "classic concurrency", and the newer API, which is normally referred to by its Java package name, `java.util.concurrent`.

In this book, we're going to talk about both approaches. In this chapter, we'll begin our journey by looking at the first of these two approaches. After that, in the next chapter, we'll introduce `java.util.concurrent`. Much later, we'll return to the subject of concurrency in Chapter 15 "Advanced Concurrent Programming" - which discusses advanced techniques, concurrency in non-Java JVM languages, and the interplay between concurrency and functional programming.

Let's get started and meet the classic approach to concurrency. This was the only API available until Java 5. As you might guess from the alternative name "synchronization-based concurrency", this is the language-level API that is built into the platform and depends upon the `synchronized` and `volatile` keywords.

It is a low-level API and can be somewhat difficult to work with, but it is very much worth understanding. It provides a solid foundation for the chapters later in the book that explain other types and aspects of concurrency.

In fact, correctly reasoning about the other forms of concurrency is very difficult without at least a working knowledge of the low-level API and concepts that we will introduce in this chapter. As we encounter the relevant topics, we will also introduce enough theory to illuminate the other views of concurrency that we'll discuss later on in the book - including when we meet concurrency in non-Java languages.

To make sense of Java's approach to concurrent programming, we're going to start off by talking about a small amount of theory.

After that, we'll discuss the impact that "design forces" have in the design and implementation of systems. We'll talk about the two most important of these forces, *safety* and *liveness*, and mention some of the others.

An important section (and the longest one in the chapter) is the detail of block-structured concurrency and an exploration of the low-level threading API.

We'll conclude this chapter by discussing the Java Memory Model, and then using the bytecode techniques that we learned in Chapter 4 to understand the real source of some common complexities in concurrent Java programming.

5.1 Concurrency theory primer

Let's get started on our journey into concurrency with a cautionary tale before we meet some basic theory.

5.1.1 But I already know about `Thread`

It's one of the most common (and potentially deadly) mistakes a developer can make — to assume that an acquaintance with `Thread`, `Runnable`, and the language-level basic primitives of Java's concurrency mechanism is enough to be a competent developer of concurrent code. In fact, the subject of concurrency is a very large one, and good multithreaded development is difficult and continues to cause problems for even the best developers with years of experience under their belts.

It is also true that the area of concurrency is undergoing a massive amount of active research at present — this has been going on for at least the last 5-10 years and shows no signs of abating. These innovations are very likely to have an impact on Java and the other languages you'll use over the course of your career.

In the first edition of this book, we made the claim that:

"If we were to pick one fundamental area of computing that's likely to change radically in terms of industry practice over the next five years, it would be concurrency."

Not only has history borne out this claim, but we feel comfortable rolling this prediction forward

- the next five years will see a continued emphasis on the different approaches to concurrency that are now part of the programming landscape.

So rather than try to be a definitive guide to every aspect of concurrent programming, the aim of this chapter is to make you aware of the underlying platform mechanisms that explain why Java's concurrency works the way it does. We'll also cover enough general concurrency theory to give you the vocabulary to understand the issues involved, and to teach you about both the necessity and the difficulty involved in getting concurrency right.

First, we'll discuss what every well-grounded Java developer should know about hardware and one of the most important theoretical limitations of concurrency.

5.1.2 Hardware

Let's start with some basic facts about concurrency and multithreading:

- Concurrent programming is fundamentally about performance.
- There are basically no good reasons for implementing a concurrent algorithm if the system you are running on has sufficient performance that a serial algorithm will do.
- Modern computer systems have multiple processing cores - even mobile phones have 2 or 4 cores today.
- All Java programs are multi-threaded, even those that only have a single application thread.

This last point is true because the JVM is itself a multi-threaded binary that can use multiple cores (e.g. for JIT compilation or garbage collection). Not only that, but the standard library also includes APIs that use *runtime-managed concurrency* to implement multi-threaded algorithms for some execution tasks.

NOTE

This means that it is entirely possible that a Java application will run faster just by upgrading the JVM it runs on, due to performance improvements in the runtime.

There will be a fuller discussion of hardware in Chapter 7, but these basic facts are so fundamental, and so relevant to concurrent programming that we want to introduce them immediately.

Now let's meet *Amdahl's Law*, named after an early IBM computer scientist, Gene Amdahl, sometimes called the "father of the mainframe".

5.1.3 Amdahl's Law

This is a simple, rough-and-ready model for reasoning about the efficiency of sharing work over multiple execution units. In the model, the execution units are abstract - so you can think of them as threads but they could also be processes, or any other entity that is capable of carrying out work.

NOTE

None of the setup or consequences to Amdahl's Law depend upon the details of how the work is done or the precise nature of the execution units, or how the computing systems are implemented.

The basic premise is that we have a single task that can be subdivided into smaller units for processing. This allows us to use multiple execution units to speed up the time taken to complete the work.

So, if we have N processors (or threads to do the work), then we might naively expect the elapsed time to be T_1 / N (if T_1 is the time the job would take on a single processor). In this model, we can finish the job as quickly as we like by just add execution units - and thereby increasing N .

However - splitting the work up is not free! There is a (hopefully small) overhead involved in the subdividing and recombination of the task. Let's assume that this *communication overhead* (sometime called the *serial part* of the calculation) is an overhead that amounts to a few percent - and we can represent it by a number s ($0 < s < 1$). So a typical value for s might be 0.05 (or 5%, whichever way you'd prefer to express it).

This means that the task will always take at least $s * T_1$ to complete - no matter how many processing units we throw at it.

This assumes that s does not depend upon N , of course, but in practice, the dividing up of work that s represents may get more complex and require *more* time as N increases. It is extremely difficult to conceive of a system architecture in which s *decreases* as N increases. So the simple assumption of "s is constant" is usually understood to be a *best-case* scenario.

So, the easiest way to think about Amdahl's Law is that:

If s is between 0 and 1 then the maximum speedup that can be achieved is $1 / s$.

This result is somewhat depressing - it means that if the communication overhead is just 2% then the maximum speedup that can ever be achieved (even with thousands of processors working at full speed) is 50X.

Amdahl's Law has a slightly more complex formulation, which is represented like this:

$$T(N) = s + (1/N) * (T_1 - s)$$

This can be seen visually in Figure 5.1 - note that the X-axis is a logarithmic scale - the convergence to $1 / s$ would be very hard to see in a linear scale representation.

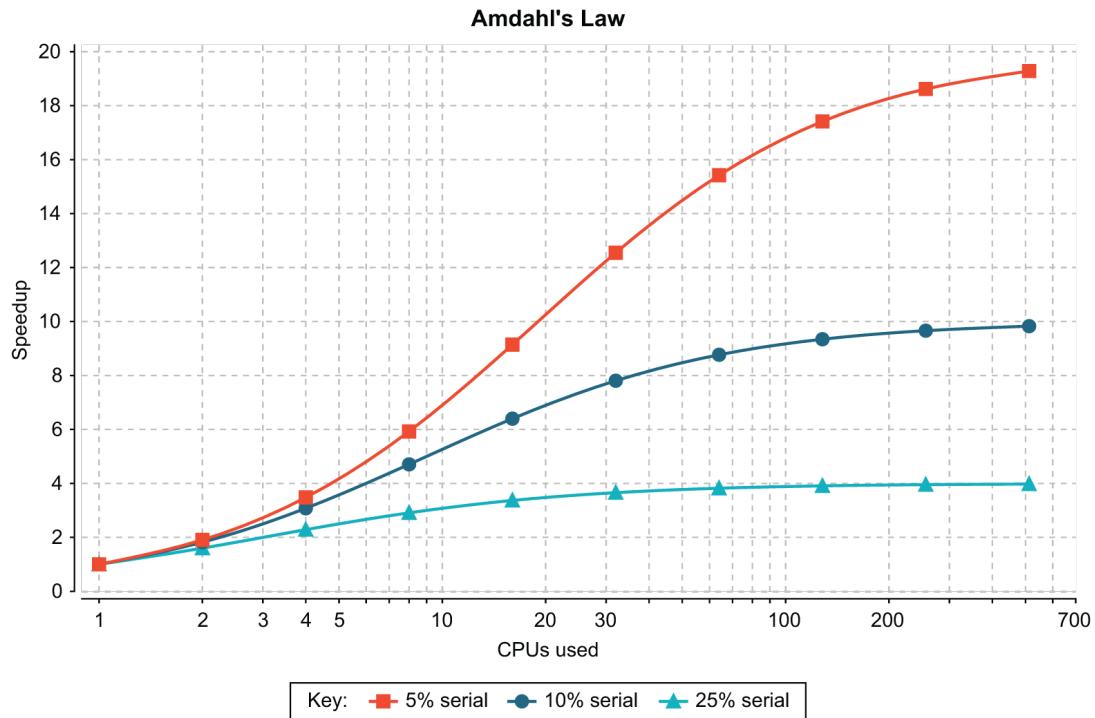


Figure 5.1 Amdahl's Law

Having set the scene with hardware and a first, very simple, concurrency model, let's dive into the specifics of how Java handles threading.

5.1.4 Explaining Java's threading model

Java's threading model is based on two fundamental concepts:

- Shared, visible-by-default mutable state
- Pre-emptive thread scheduling by the operating system

Let's consider some of the most important aspects of these ideas:

- Objects can be easily shared between all threads within a process.
- Objects can be changed ("mutated") by any threads that have a reference to them.
- The thread scheduler (the operating system) can swap threads on and off cores at any time, more or less.
- Methods must be able to be swapped out while they're running (otherwise a method with an infinite loop would steal the CPU forever).
- This, however, runs the risk of an unpredictable thread swap leaving a method "half-done" and an object in an inconsistent state.
- Objects can be *locked* to protect vulnerable data.

The last point is absolutely crucial - without it there is a huge risk of changes being made in one thread not being seen correctly in other threads. In Java, the ability to lock objects is provided by the `synchronized` keyword in the core language.

NOTE

Technically, Java provides monitors on each of its objects, which combine a lock (aka mutual exclusion) with the ability to wait for a certain condition to become true.

Java's thread and lock-based concurrency is very low-level, and often hard to work with. To cope with this, a set of concurrency libraries, known as `java.util.concurrent` after the Java package where the new classes live, was introduced in Java 5. This provided a set of tools for writing concurrent code that many programmers find easier to use than the classic block-structured concurrency primitives.

We will discuss `java.util.concurrent` in the next chapter and will focus on the language-level API for now.

5.1.5 Lessons learned

Java was the first mainstream programming language to have built-in support for multithreaded programming. This represented a huge step forward at the time, but now, 15 years later, we've learned a lot more about how to write concurrent code.

It turns out that some of Java's initial design decisions are quite difficult for most programmers to work with. This is unfortunate, because the increasing trend in hardware is toward processors with many cores, and the only good way to take advantage of those cores is with concurrent code. We'll discuss some of the difficulties of concurrent code in this chapter. The subject of modern processors naturally requiring concurrent programming is covered in some detail in chapter 7 where we discuss performance.

As developers become more experienced with writing concurrent code, they find themselves running up against recurring concerns that are important to their systems. We call these concerns *design forces*. They're high-level concepts that exist (and often conflict) in the design of practical concurrent OO systems.

We're going to spend a little bit of time looking at some of the most important of these forces in the next couple of sections.

5.2 Design concepts

The most important design forces were catalogued by Doug Lea as he was doing his landmark work producing `java.util.concurrent`:

- Safety (also known as *concurrent type safety*)
- Liveness
- Performance
- Reusability

Let's look at each of these forces now.

5.2.1 Safety and concurrent type safety

Safety is about ensuring that object instances remain self-consistent regardless of any other operations that may be happening at the same time. If a system of objects has this property, it's said to be *safe* or *concurrently typesafe*.

As you might guess from the name, one way to think about concurrency is in terms of an extension to the regular concepts of object modeling and type safety. In non-concurrent code, you want to ensure that regardless of what public methods you call on an object, it's in a well-defined and consistent state at the end of the method. The usual way to do this is to keep all of an object's state private and expose a public API of methods that only alter the object's state in way that makes sense for the design domain.

Concurrent type safety is the same basic concept as type safety for an object, but applied to the much more complex world in which other threads are potentially operating on the same objects on different CPU cores at the same time.

For example, consider this simple class:

```
public class StringStack {
    private String[] values = new String[16];
    private int current = 0;

    public boolean push(String s) {
        if (current < values.length) {
            values[current] = s;
            current = current + 1;
        }
        return false;
    }

    public String pop() {
        if (current < 1) {
            return null;
        }
        current = current - 1;
        return values[current];
    }
}
```

When used by single-threaded client code, this is fine. However, pre-emptive thread scheduling can cause problems. For example, a context switch can occur at this point in the code:

```
public boolean push(String s) {
    if (current < values.length) {
        values[current] = s;
        // .... context switch here
        current = current + 1;
    }
    return false;
}
```

①

- ① The object is left in an inconsistent and incorrect state

If the object is then viewed from another thread, then one part of the state (`values`) will have been updated but the other (`current`) will not.

Exploring, and solving, this problem is the primary theme of this chapter.

In general, one strategy for safety is to never return from a non-private method in an inconsistent state, and to never call any non-private method (and certainly not a method on any other object) while in an inconsistent state. If this is combined with a way of protecting the object (such as a synchronization lock or critical section) while it's inconsistent, the system can be guaranteed to be safe.

5.2.2 Liveness

A live system is one in which every attempted activity eventually either progresses or fails. A system that is not live is basically stuck - it will neither progress towards success or fail.

The key word in the definition is *eventually* — there is a distinction between a transient failure to progress (which isn't that bad in isolation, even if it's not ideal) and a permanent failure. Transient failures could be caused by a number of underlying problems, such as:

- Locking or waiting to acquire a lock
- Waiting for input (such as network I/O)
- Temporary failure of a resource
- Not enough CPU time available to run the thread

Permanent failures could be due to a number of causes. These are some of the most common:

- Deadlock
- Unrecoverable resource problem (such as if the NFS goes away)
- Missed signal

We'll discuss locking and several of these other problems later in the chapter, although you may already be familiar with some or all of them.

5.2.3 Performance

The performance of a system can be quantified in a number of different ways. In chapter 7, we'll talk about performance analysis and techniques for tuning, and we'll introduce a number of other metrics you should know about. For now, think of performance as being a measure of how much work a system can do with a given amount of resources.

5.2.4 Reusability

Reusability forms a fourth design force, because it isn't really covered by any of the other considerations. A concurrent system that has been designed for easy reuse is sometimes very desirable, although this isn't always easy to implement. One approach is to use a reusable toolbox (like `java.util.concurrent`) and build non-reusable application code on top of it.

5.2.5 How and why do the forces conflict?

The design forces are often in opposition to each other, and this tension can be viewed as a central reason why designing good concurrent systems is difficult.

- Safety stands in opposition to liveness — safety is about ensuring that bad things don't happen, whereas liveness requires progress to be made.
- Reusable systems tend to expose their internals, which can cause problems with safety.
- A naïvely written safe system will typically not be very performant, as it usually resorts to the heavy use of locking to provide safety guarantees.

The balance that you should ultimately try to achieve is for the code to be flexible enough to be useful for a wide range of problems, closed enough to be safe, and still reasonably live and performant. This is quite a tall order, but, fortunately, there are some practical techniques to help with this. Here are some of the most common in rough order of usefulness:

1. Restrict the external communication of each subsystem as much as possible. Data hiding is a powerful tool for aiding with safety.
2. Make the internal structure of each subsystem as deterministic as possible. For example, design in static knowledge of the threads and objects in each subsystem, even if the subsystems will interact in a concurrent, nondeterministic way.
3. Apply policy approaches that client apps must adhere to. This technique is powerful, but relies on user apps cooperating, and it can be hard to debug if a badly behaved app disobeys the rules.
4. Document the required behavior. This is the weakest of the alternatives, but it's sometimes necessary if the code is to be deployed in a very general context.

The developer should be aware of each of these possible safety mechanisms and should use the strongest possible technique, while being aware that there are circumstances in which only the weaker mechanisms are possible.

5.2.6 Sources of overhead

There are many aspects of a concurrent system that can contribute to the inherent overhead:

- Monitors (i.e. locks and condition variables)
- Number of context switches
- Number of threads
- Scheduling
- Locality of memory
- Algorithm design

This should form the basis of a checklist in your mind. When developing concurrent code, you should ensure that you have thought about everything on this list.

In particular, the last of these - algorithm design - is an area in which developers can really distinguish themselves, because learning about algorithm design will make you a better programmer in any language.

Two standard texts (highly recommended by the authors) are *Introduction to Algorithms* by Cormen et al. (MIT, 2009) — don't be deceived by the title, this is a serious work — and *The Algorithm Design Manual*, second edition, by Skiena (Springer-Verlag, 2008). For both single-threaded and concurrent algorithms, these are excellent choices for further reading.

We'll mention many of these sources of overhead in this chapter and the subsequent ones (especially chapter 7, about performance), but now let's turn to our next subject — a review of Java's "classic" concurrency and a close look at why programming with it can be difficult.

5.3 Block-structured concurrency (pre-Java 5)

Much of our coverage of Java concurrency is about discussing alternatives to the language-level, aka block-synchronization-based, aka *intrinsic* approach to concurrency. But to get the most out of the discussion of the alternatives, it's important to have a firm grasp of what's good and bad about the classic view of concurrency.

To that end, for the rest of this chapter, we'll discuss the original, quite low-level way of tackling multithreaded programming using Java's concurrency keywords — `synchronized`, `volatile`, and so on. This discussion will take place in the context of the design forces and with an eye to what will come later on.

Following on from that, we'll briefly consider the lifecycle of a thread, and then discuss common techniques (and pitfalls) of concurrent code, such as fully synchronized objects, deadlocks, the `volatile` keyword, and immutability.

Let's get started with an overview of synchronization.

5.3.1 Synchronization and locks

As you probably already know, the `synchronized` keyword can be applied either to a block or to a method. It indicates that before entering the block or method, a thread must acquire the appropriate lock. For example, let's think about a method to withdraw money from a bank account:

```
public synchronized boolean withdraw(final int amount) {    ①
    // Check to see amount > 0, throw if not
    if (balance >= amount) {
        balance = balance - amount;
        return true;
    }
    return false;
}
```

- ① Only one thread can try to withdraw from this account at once.

For a method, that means acquiring the lock belonging to the object instance (or the lock belonging to the `Class` object for `static synchronized` methods). For a block, the programmer should indicate which object's lock is to be acquired.

Only one thread can be progressing through any of an object's synchronized blocks or methods at once; if other threads try to enter, they're suspended by the JVM. This is true regardless of whether the other thread is trying to enter either the same or a different synchronized block on the same object. In concurrency theory, this type of construct is sometimes referred to as a *critical section* - but this term is more commonly used in C++ than in Java.

NOTE

Have you ever wondered why the Java keyword used for a critical section is `synchronized`? Why not "critical" or "locked"? What is it that's being synchronized? We'll return to this in section 4.2.5, but if you don't know or have never thought about it, you may want to take a couple of minutes to ponder it before continuing.

Let's look at some basic facts about synchronization and locks in Java. Hopefully you already have most (or all) of these at your fingertips:

- Only objects — not primitives — can be locked.
- Locking an array of objects doesn't lock the individual objects.
- A synchronized method can be thought of as equivalent to a `synchronized (this) { ... }` block that covers the entire method (but note that they're represented differently in bytecode).
- A static synchronized method locks the `Class` object, because there's no instance object to lock.
- If you need to lock a class object, consider carefully whether you need to do so explicitly, or by using `getClass()`, because the behavior of the two approaches will be different in a subclass.

- Synchronization in an inner class is independent of the outer class (to see why this is so, remember how inner classes are implemented).
- `synchronized` doesn't form part of the method signature, so it can't appear on a method declaration in an interface.
- Unsynchronized methods don't look at or care about the state of any locks, and they can progress while synchronized methods are running.
- Java's locks are reentrant. That means a thread holding a lock that encounters a synchronization point for the same lock (such as a `synchronized` method calling another `synchronized` method on the same object) will be allowed to continue.

NOTE

Non-reentrant locking schemes do exist in other languages (and can be synthesized in Java — see the detail of the Javadoc for `ReentrantLock` in `java.util.concurrent.locks` if you want the gory details) but they're generally painful to deal with, and they're best avoided unless you really know what you're doing.

That's enough review of Java's synchronization. Now let's move on to discuss the states that a thread moves through during its lifecycle.

5.3.2 The state model for a thread

In Figure 5.2, you can see how a thread lifecycle progresses—from creation to running, to possibly being suspended, before running again (or blocking on a resource), and eventually completing.

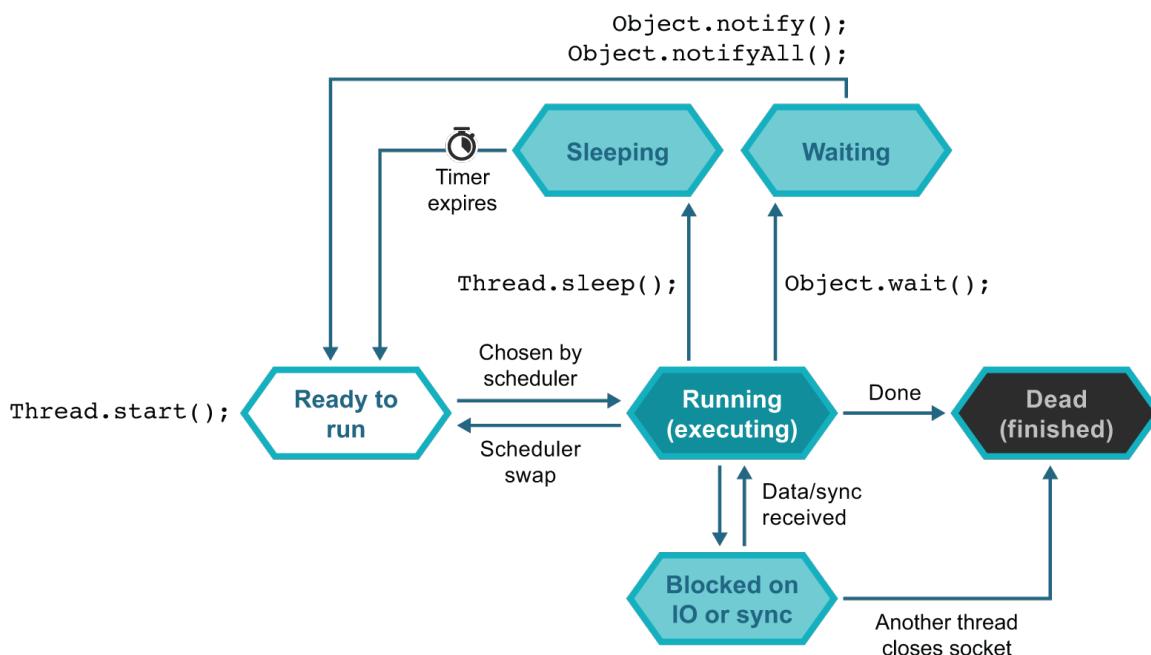


Figure 5.2 The state model of a Java thread

A thread is initially created in the Ready state. The scheduler will then find a core for it to run upon, and some small amount of waiting time may be involved if the machine is heavily loaded. From there, the thread will usually consume its time allocation and be placed back into the Ready state to await further processor time slices. This is the action of the forcible thread scheduling that we mentioned in section 4.1.1.

As well as the standard action by the scheduler, the thread itself can indicate that it isn't able to make use of the core at this time. This can be achieved in two different ways:

1. The program code indicates that the thread should wait for a fixed time before continuing by calling `Thread.sleep()`
2. The thread must wait until notified that some external condition has been met, by calling `Object.wait()`

In both cases, the thread is immediately removed from the core. However, the behavior after that is different in both cases.

In the first case, the thread is sleeping for a definite amount of time. The operating system sets a timer, and when it expires, the sleeping thread is woken up and is ready to run again, and is placed back in the Ready state.

The second case is slightly different. It uses the condition aspect of Java's per-object monitors. The thread will wait indefinitely, and will not normally wake up until the operating system signals that the condition may have been met usually by some other thread calling `Object.notify()` on the current object).

As well as these two possibilities that are under the threads control, a thread can be blocked because it's waiting on I/O or to acquire a lock held by another thread. In this case, the thread isn't swapped off the core but is kept busy, waiting for the lock or data to become available. If this happens, the thread will continue to execute until the end of its timeslice.

Let's move on to talk about one well-known way to solve the synchronization problem. This is the idea of fully synchronized objects.

5.3.3 Fully synchronized objects

Earlier in this chapter, we introduced the concept of concurrent type safety and mentioned one strategy for achieving this. Let's look at a more complete description of this strategy, which is usually called *fully synchronized objects*. If all of the following rules are obeyed, the class is known to be thread-safe and will also be live.

A fully synchronized class is a class that meets all of these conditions:

- All fields are always initialized to a consistent state in every constructor.
- There are no public fields.
- Object instances are guaranteed to be consistent after returning from any non-private method (assuming the state was consistent when the method was called).
- All methods provably terminate in bounded time.
- All methods are synchronized.
- No method calls another instance's methods while in an inconsistent state.
- No method calls any non-private method on the current instance while in an inconsistent state.

Listing 5.1 shows an example of such a class from the backend of a banking system. The class `FSOAccount` models an account - the FSO prefix is there to clearly indicate that this implementation uses full-synchronized objects.

This situation provides both deposits, withdrawals and balance queries - a classic conflict between read and write operations, so synchronization is used to prevent inconsistency.

Listing 5.1 A fully synchronized class

```
public class FSOAccount {
    private double balance;                                ①

    public FSOAccount(double openingBalance) {
        // Check to see openingBalance > 0, throw if not
        balance = openingBalance;                          ②
    }

    public synchronized boolean withdraw(final int amount) { ③
        // Check to see amount > 0, throw if not
        if (balance >= amount) {
            balance = balance - amount;
            return true;
        }
        return false;
    }

    public synchronized void deposit(final int amount) {      ③
        // Check to see amount > 0, throw if not
        balance = balance + amount;
    }

    public synchronized double getBalance() {                ③
        return balance;
    }
}
```

- ① No public fields
- ② All fields initialized in constructor
- ③ All methods are synchronized

This seems fantastic at first glance — the class is both safe and live. The problem comes with performance — just because something is safe and live doesn't mean it's necessarily going to be very quick. You have to use synchronized to coordinate all the accesses (both get and put) to the balance, and that locking is ultimately going to slow you down. This is a central problem of this way of handling concurrency.

In addition to the performance problems, the code in listing 5.1 is quite fragile. You can see that you never touch balance outside of a synchronized method, but this is only possible to check by eye due to the small amount of code in play.

In real, larger systems, this sort of manual verification would not be possible due to the amount of code. It's too easy for bugs to creep into larger codebases that use this approach, which is another reason that the Java community began to look for more robust approaches.

5.3.4 Deadlocks

Another classic problem of concurrency (and not just Java's take on it) is the *deadlock*. Consider listing 5.2, which is a slightly extended form of the last example. In this version, as well as modeling the account balance, we also have a transferTo() method that can move money from one account to another.

NOTE

This is a naïve attempt to build a multithreaded transaction system. It's designed to demonstrate deadlocking — you shouldn't use this as the basis for real code.

In Listing 5.2 let's add a method to transfer funds between two FSOAccount objects, like this:

Listing 5.2 deadlocking example

```
public synchronized boolean transferTo(FSOAccount other, int amount) {
    // Check to see amount > 0, throw if not
    // Simulate some other checks that need to occur
    try {
        Thread.sleep(10);
    } catch (InterruptedException __) {
    }
    if (balance >= amount) {
        balance = balance - amount;
        other.deposit(amount);
        return true;
    }
    return false;
}
```

Now, let's actually introduce some concurrency in a main class:

```

public class FSOMain {
    private static final int MAX_TRANSFERS = 1_000;

    public static void main(String[] args) throws InterruptedException {
        FSOAccount a = new FSOAccount(10_000);
        FSOAccount b = new FSOAccount(10_000);
        Thread tA = new Thread(() -> {
            for (int i = 0; i < MAX_TRANSFERS; i = i + 1) {
                boolean ok = a.transferTo(b, 1);
                if (!ok) {
                    System.out.println("Thread A failed at " + i);
                }
            }
        });
        Thread tB = new Thread(() -> {
            for (int i = 0; i < MAX_TRANSFERS; i = i + 1) {
                boolean ok = b.transferTo(a, 1);
                if (!ok) {
                    System.out.println("Thread B failed at " + i);
                }
            }
        });
        tA.start();
        tB.start();
        tA.join();
        tB.join();

        System.out.println("End: " + a.getBalance() + " : " + b.getBalance());
    }
}

```

At first glance, this code looks sensible. You have two transactions being performed by separate threads. This doesn't seem too outlandish a design — just threads sending money between the two accounts - and all the methods are synchronized.

Note that we've introduced a small sleep into the `transferTo()` method. This is to allow the thread scheduler to run both threads and lead to the possibility of deadlock.

NOTE

The sleep is for demonstration purposes, not because it is something you'd actually do when writing code for a bank transfer. It's there to simulate code that would actually be there in practice - a delay caused by a call to a database or an authorization check.

If you run the code, you'll normally see an example of a deadlock — both threads will run for a bit and eventually get stuck.

The reason for this is that each thread requires the other to release the lock it holds before the transfer method can progress. This can be seen in Figure 5.3.

Threads need both x and y to progress

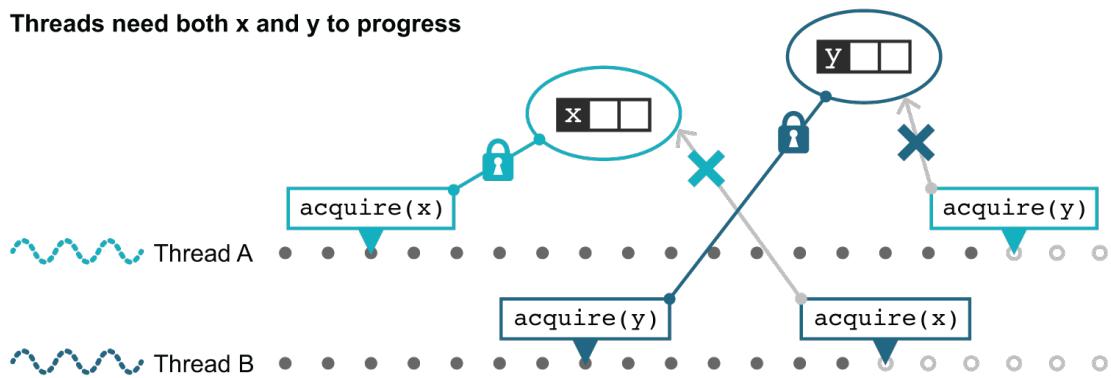


Figure 5.3 Deadlocked threads

Another way of looking at this can be seen in figure 5.4, where we show the Thread Dump view from the JDK Mission Control tool (we will have more to say about this tool in Chapter 7, and will show you how to find this useful view then).

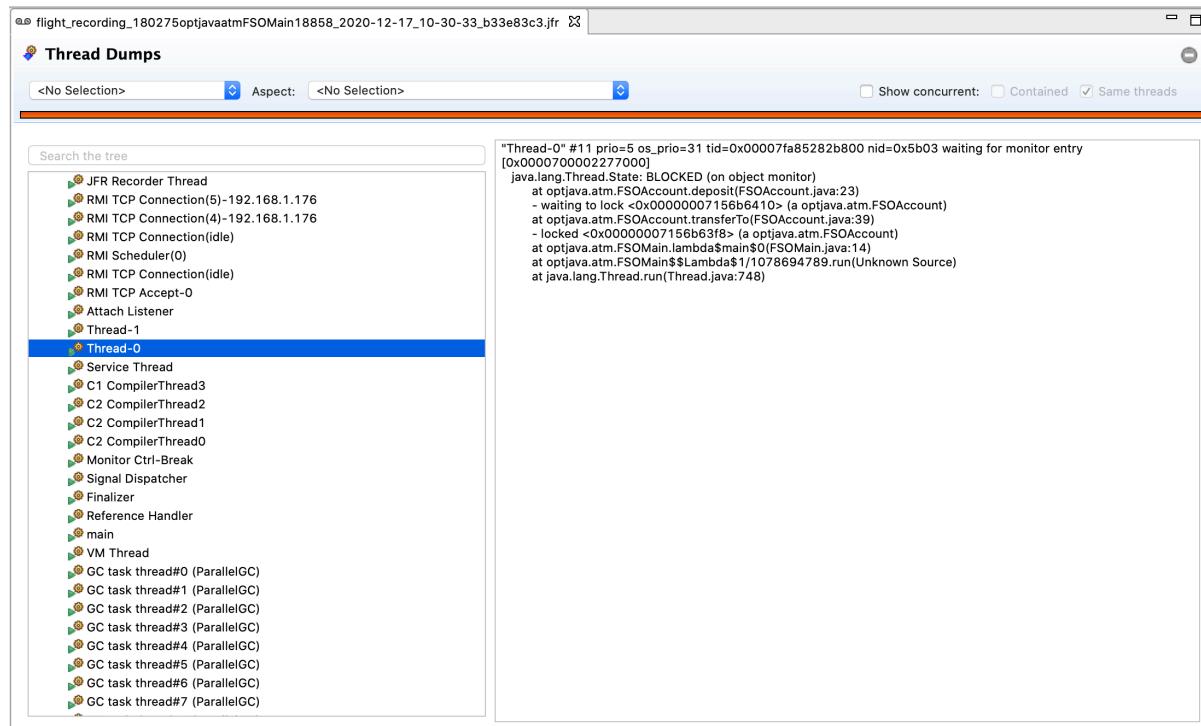


Figure 5.4 Deadlocked threads

The two threads have been created as Thread-0 and Thread-1 and we can see that Thread-0 has locked one referenced and is BLOCKED waiting to lock the other. The corresponding thread dump for Thread-1 would show the opposite configuration of the locks - hence the deadlock.

NOTE

In terms of the fully synchronized object approach, this deadlock occurs because of the violation of the "bounded time" principle. When the code calls `other.deposit()` we cannot guarantee how long the code will run for, because the Java Memory Model gives us no guarantees on when a blocked monitor will be released.

To deal with deadlocks, one technique is to always acquire locks in the same order in every thread. In the preceding example, the first thread to start acquires them in the order A, B, whereas the second thread acquires them in the order B, A. If both threads had insisted on acquiring in order A, B, the deadlock would have been avoided, because the second thread would have been blocked from running at all until the first had completed and released its locks.

Later on in the chapter, we will show a simple way to arrange for all locks to be obtained in the same order and a way to verify that this is indeed satisfied.

Next, we'll return to a puzzle we posed earlier: why the Java keyword for a critical section is named `synchronized`. This will then lead us into a discussion of the `volatile` keyword.

5.3.5 Why `synchronized`?

A simple conceptual model of concurrent programming is timesharing of a CPU — that is threads swapping on and off a single core. This classic view is shown in Figure 5.5

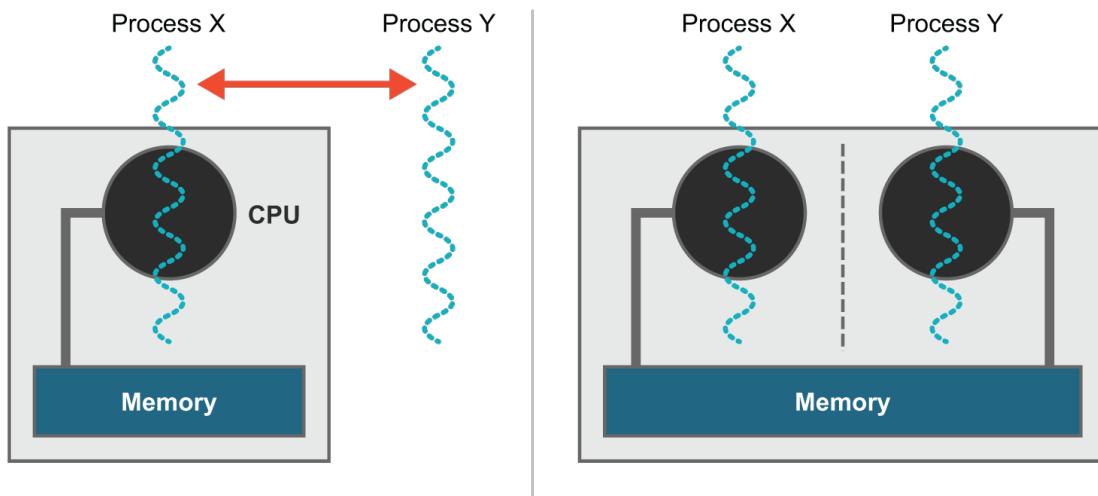


Figure 5.5 Single (left) and multi (right) core of thinking about concurrency and threads

However, this has not been an accurate picture of modern hardware for many years now.

Twenty years ago, a working programmer could go for years on end without encountering a system that had more than one or at most two processing cores. That is no longer the case.

Today, anything as big as or larger than a mobile phone has multiple cores, so the mental model should be different too, encompassing multiple threads all running on different cores at the same physical moment (and potentially operating on shared data).

You can see this in figure 5.5. For efficiency, each thread that is running simultaneously may have its own cached copy of data being operated on.

NOTE

We will still present theoretical models of execution where our hypothetical computer has only one core. This is purely so that you can see that the non-deterministic concurrency problems we are discussing are inherent, and not caused by particular aspects of hardware design.

With this picture in mind, let's turn to the question of the choice of keyword used to denote a locked section or a method.

We asked earlier, what is it that's being synchronized in the code in listing 4.1? The answer is: *The memory representation in different threads of the object being locked is what is being synchronized.* That is, after the synchronized method (or block) has completed, any and all changes that were made to the object being locked are flushed back to main memory before the lock is released, as illustrated in figure 5.6.

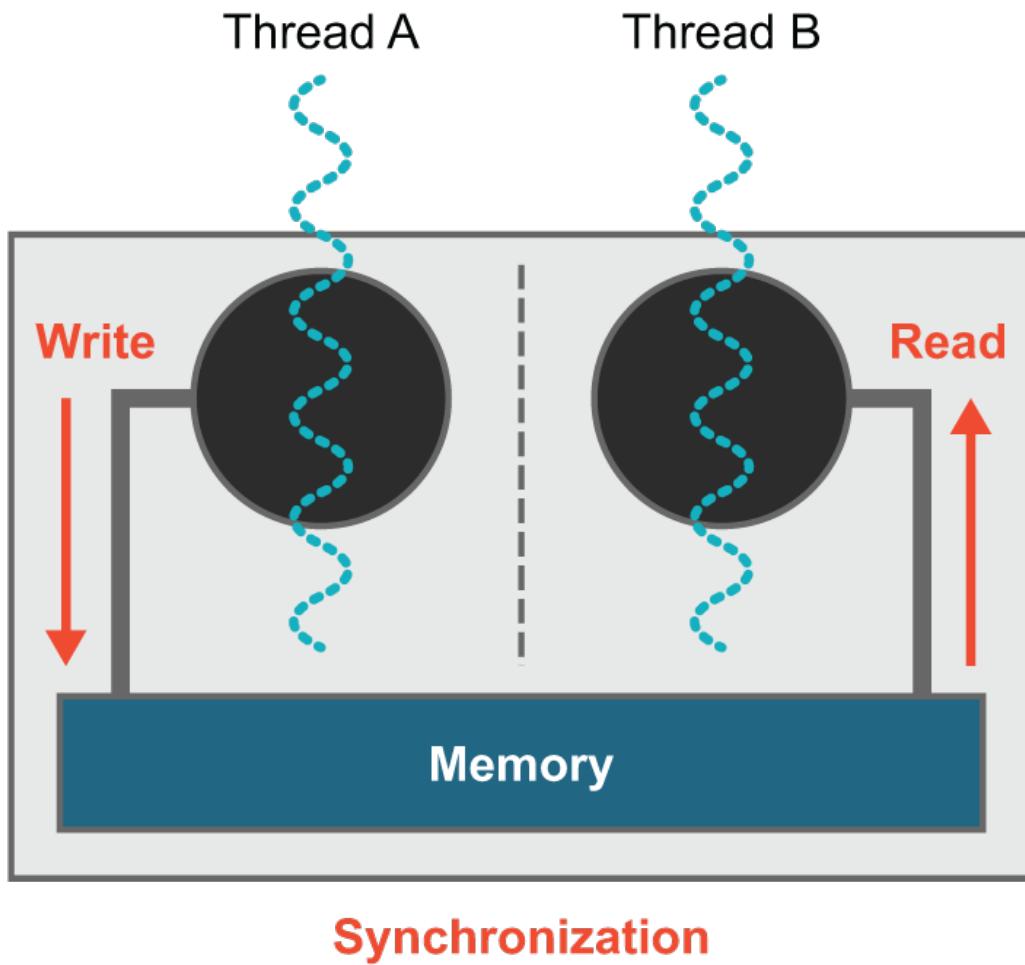


Figure 5.6 A change to an object propagates between threads via main memory

In addition, when a synchronized block is entered, then after the lock has been acquired, any changes to the locked object are read in *from* main memory, so the thread with the lock is synchronized to main memory's view of the object before the code in the locked section begins to execute.

5.3.6 The volatile keyword

Java has had the `volatile` keyword since the dawn of time (Java 1.0), and it's used as a simple way to deal with concurrent handling of object fields, including primitives. The following rules govern a `volatile` field:

- The value seen by a thread is always re-read from main memory before use.
- Any value written by a thread is always flushed through to main memory before the bytecode instruction completes.

This is sometimes described as being "like a tiny little synchronized block" around the single

operation, but this is misleading because `volatile` does not involve any locking.

The action of `synchronized` is to use a mutual exclusion lock on an object to ensure that only one thread can execute a synchronized method on that object. Synchronized methods can contain many read and operations on the object and they will be executed as an indivisible unit (from the point of view of other threads) because the results of the method executing on the object are not seen until the method exits and the object is flushed back to main memory.

The key point about `volatile` is that it allows for only *one* operation on the memory location, which will be immediately flushed to memory. This means either a single read, *or* a single write - but not more than that. We saw these two sorts of operations in Figure 5.6

A volatile variable should only be used to model a variable where writes to the variable don't depend on the current state (the read state) of the variable. This is a consequence of `volatile` only guaranteeing a single operation.

For example, the `++` and `--` operators are not safe to use on a volatile, as they are equivalent to `v = v + 1` or `v = v - 1`. The increment example is a classic example of a *state-dependent update*

For cases where the current state matters, you must always introduce a lock to be completely safe.

So, `volatile` allows the programmer to write simplified code in some cases, but at the cost of the extra flushes on every access. Notice also that because the volatile mechanism doesn't introduce any locks, you can't deadlock using volatiles - only with synchronization.

Later in the chapter, we will meet some other applications for `volatile` and discuss the mechanism in more detail.

5.3.7 Thread states and methods

A `java.lang.Thread` object is just that - a Java object that lives in the heap, and contains metadata about an operating system thread that either exists, used to exist, or will potentially exist in the future.

Java defines these states for a thread object, which correspond to OS thread states on mainstream operating systems:

- NEW - Thread object has been created but the actual OS thread has not
- RUNNABLE - Thread is available to run. OS is responsible for scheduling it.
- BLOCKED - Thread is not running - it needs to acquire a lock or is in a system call
- WAITING - Thread is not running - it has called `Object.wait()` or `Thread.join()`
- TIMED_WAITING - Thread is not running - it has called `Thread.sleep()`
- TERMINATED - Thread is not running - it has completed execution.

All threads start in the NEW state and finish in the TERMINATED state, whether the thread's `run()` method exits normally or throws an exception.

NOTE

The Java thread state model does not distinguish between whether a RUNNABLE thread is actually physically executing at that precise moment, or is waiting (in the run queue).

The actual creation of the thread is done by the `start()` method, which calls into native code to actually perform the relevant system calls (e.g. `clone()` on Linux) to create the thread and begin code execution in the thread's `run()` method.

The standard thread API in Java breaks down into three groups of methods. Rather than include a lot of boilerplate javadoc descriptions of each method, we will just list them and leave the reader to consult the API docs for more detail.

The first is a group of methods for reading metadata about the thread:

- `getId()`
- `getName()`
- `getState()`
- `getPriority()`
- `isAlive()`
- `isDaemon()`
- `isInterrupted()`

Some of this metadata (such as the thread id obtained from `getId()`) will be fixed for the lifetime of the thread. Some of it, such as the thread state and interrupted state will naturally change as the thread runs, and some of it (e.g. name and daemon status) can be set by the programmer. This leads us to the second group of methods:

- `setDaemon()`
- `setName()`
- `setPriority()`
- `setUncaughtExceptionHandler()`

It is often better for the programmer to configure any appropriate properties for threads before starting them.

Finally, there is a set of thread control methods, that are used to start new threads and interact with other running threads:

- `start()`
- `interrupt()`
- `join()`

Note that `Thread.sleep()` does not appear in this list, because it's a static method that only targets the current thread.

NOTE

Some of the thread methods with timeouts (e.g. `Thread.join()` with a timeout parameter) may actually result in the thread being placed into `TIMED_WAITING` instead of `WAITING`

Let's take a look at an example of how to use the thread methods in a typical lifecycle of a simple multithreaded application:

```
Runnable r = () -> {
    var start = System.currentTimeMillis();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    var thisThread = Thread.currentThread();
    System.out.println(thisThread.getName() +
        " slept for " + (System.currentTimeMillis() - start));
};

var t = new Thread(r);                                ①
t.setName("Worker");
t.start();                                         ②
Thread.sleep(100);
t.join();                                           ③
System.out.println("Exiting");
```

- ① Thread metadata object created
- ② Operating system creates actual thread
- ③ Main thread pauses and waits for worker to exit before continuing

This is pretty simple stuff - the main thread creates the worker, starts it, then waits for at least 100ms (to give the scheduler a chance to run) before reaching the `join()` call, which causes it to pause until the worker thread exits.

In the meantime, the worker thread completes the sleep, wakes up again, and prints out the message.

NOTE

The elapsed time for the sleep will most likely not be exactly 1000ms. The operating system scheduler is non-deterministic and so the best guarantee that is offered is that the operating system will attempt to ensure that the thread sleeps for the requested amount of time, unless woken.

However, multithreaded programming is often about dealing with unexpected circumstances, as we will see in the next section.

WORKING WITH EXCEPTIONS AND THREADS

When working with threads it is often necessary to distinguish those operations that could potentially be interrupted. Such methods, such as the infamous `Thread.sleep()`, are sometimes a source of annoyance to the programmer, as they require the `InterruptedException` checked exception to be handled.

So what happens if one of these methods is actually interrupted? Let's look at a slightly modified version of the previous example:

```
Runnable r = () -> {
    var start = System.currentTimeMillis();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) { ③
        e.printStackTrace();
    }
    var thisThread = Thread.currentThread();
    System.out.println(thisThread.getName() +
        " slept for " + (System.currentTimeMillis() - start));
    if (thisThread.isInterrupted()) {
        System.out.println("Thread " + thisThread.getName() + " interrupted");
    }
};

var t = new Thread(r);
t.setName("Worker"); ①
t.start();
Thread.sleep(100);
t.interrupt(); ②
t.join();
System.out.println("Exiting");
```

- ① Creates worker thread
- ② Main thread interrupts worker and wakes it up
- ③ Worker thread is interrupted and execution continues here

When we run this code, we see some output like this:

```
java.lang.InterruptedException: sleep interrupted
    at java.base/java.lang.Thread.sleep(Native Method)
    at examples.LifecycleWithInterrupt.lambda$main$0
        (LifecycleWithInterrupt.java:9)
    at java.base/java.lang.Thread.run(Thread.java:832)
Worker slept for 101
Exiting
```

If you look closely, however, you will see that the message:

"Thread Worker interrupted"

does not appear. This is because, when the worker thread is interrupted, it causes an exception to

be thrown - but it also clears the interrupt status. The only difference between a thread that has been interrupted and not is that the catch block executes.

If we want to be able to distinguish between a blocking call that was interrupted vs one that ran to completion, then we can do something like this:

```
Runnable r = () -> {
    var start = System.currentTimeMillis();
    var wasInterrupted = false;
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        wasInterrupted = true;
        e.printStackTrace();
    }
    var thisThread = Thread.currentThread();
    System.out.println(thisThread.getName() +
        " slept for " + (System.currentTimeMillis() - start));
    if (wasInterrupted) {
        System.out.println("Thread " + thisThread.getName() + " interrupted");
    }
};

var t = new Thread(r);
t.setName("Worker");
t.start();
Thread.sleep(100);
t.interrupt();
t.join();
System.out.println("Exiting");
```

- ➊ Set up the state to record a possible interruption
- ➋ Record the interruption

Another issue for multithreaded programming is how to handle exceptions that may be thrown from within a thread. For example, suppose that we are executing a `Runnable` of unknown provenance. If it throws an exception and dies then other code may not be aware of it.

Fortunately, the Thread API provides the ability to add uncaught exception handlers to a thread before starting it. Like this example:

```
var badThread = new Thread(() -> {
    throw new UnsupportedOperationException(); });

// Set a name before starting the thread
badThread.setName("An Exceptional Thread");

// Set the handler
badThread.setUncaughtExceptionHandler((t, e) -> {
    System.err.printf("Thread %d '%s' has thrown exception " +
        "%s at line %d of %s",
        t.getId(),
        t.getName(),
        e.toString(),
        e.getStackTrace()[0].getLineNumber(),
        e.getStackTrace()[0].getFileName());

badThread.start();
```

The handler is an instance of `UncaughtExceptionHandler` - which is a functional interface, defined like this:

```
public interface UncaughtExceptionHandler {
    void uncaughtException(Thread t, Throwable e);
}
```

This method provides a simple callback to allow thread control code to take action based on the observed exception - for example a thread pool may restart a thread that has exited in this way to maintain pool size.

NOTE

Any exception thrown by `uncaughtException()` will be ignored by the JVM

Before we move on, we need to discuss some other control methods of `Thread` which are deprecated, and which should not be used by application programmers.

DEPRECATED THREAD METHODS

Java was the first mainstream language to support multithreaded programming out of the box. However, this "first mover" advantage had its dark side - many of the inherent issues that exist with concurrent programming were first encountered by programmers working in Java.

One aspect of this is the unfortunate fact that some of the methods in the original thread API are simply unsafe to use and unfit for purpose. In particular:

```
`Thread.stop()`
```

This method is essentially impossible to use safely - it kills another thread without any warning and with no way for the killed thread to ensure that any locked objects are made safe.

The deprecation of `stop()` followed close on the heels of its active use in early Java because stopping another thread required injecting an exception into the other thread's execution. However, it's impossible to know precisely *where* that other thread is in execution. Maybe the thread is killed in the middle of a `finally` block the developer assumed would run fully and the program is left in a corrupted state.

The mechanism is that an unchecked `ThreadDeath` exception is triggered on the killed thread. It is not feasible for code to guard against such an exception with try blocks (any more than it is possible to reliably protect against an `OutOfMemoryError`) and so the exception immediately starts unwinding the stack of the killed thread, and all monitors are unlocked. This immediately makes potentially damaged objects visible to other threads and so `stop()` is just not safe for use.

As well as the reasonably well-known issues with `stop()`, several other methods also have serious issues. For example, `suspend()` does not cause any monitors to be released. This means

that any thread that attempts to access any synchronized code that is locked by the suspended thread will block permanently, unless the suspended thread is reactivated.

This represents a major liveness hazard and so `suspend()` and `resume()` should never be used. The `destroy()` method was never implemented, but it would have suffered from the same issues if it had been.

NOTE

These dangerous thread methods have been deprecated since Java 1.2 - over 20 years ago - and have recently been marked for removal (which will be a breaking change, to give you some idea of how seriously this is viewed).

The real solution to the problem of reliably controlling threads from other threads is best illustrated by the *Volatile Shutdown* pattern that we will meet later in the chapter.

Now, let's move on to one of the most useful techniques when handling data that must be shared safely when programming in a concurrent style.

5.3.8 Immutability

One technique that can be of great value is the use of immutable objects. These are objects that either have no state, or that have only final fields (which must therefore be populated in the constructors of the objects). These are always safe and live, because their state can't be mutated, so they can never be in an inconsistent state.

One problem is that any values that are required to initialize a particular object must be passed into the constructor. This can lead to unwieldy constructor calls, with many parameters. Alternatively, many coders use a *factory method* instead. This can be as simple as using a static method on the class, instead of a constructor, to produce new objects. The constructors are usually made protected or private, so that the static factory methods are the only way of instantiating.

For example, consider a simple deposit class that we might see in a banking system:

```

public final class Deposit {
    private final double amount;
    private final LocalDate date;
    private final Account payee;

    private Deposit(double amount, LocalDate date, Account payee) {
        this.amount = amount;
        this.date = date;
        this.payee = payee;
    }

    public static Deposit of(double amount, LocalDate date, Account payee) {
        return new Deposit(amount, date, payee);
    }

    public static Deposit of(double amount, Account payee) {
        return new Deposit(amount, LocalDate.now(), payee);
    }
}

```

This has the fields of the class, a private constructor and two factory methods, one of which is a convenience method for creating deposits for today.

Next up are the accessor methods for the fields:

```

public double amount() {
    return amount;
}

public LocalDate date() {
    return date;
}

public Account payee() {
    return payee;
}

```

Note that in our example, these are presented in *record style* - where the name of the accessor method matches the name of the field. This is in contrast to *bean style*, when getter methods are prefixed with `get` and setter methods (for any non-final fields) are prefixed with `set`.

Immutable objects obviously can't be changed, so what happens when we want to make changes to one of them? For example, it's very common that if a deposit or other transaction can't take place on a specific day, that the transaction is "rolled" to the following day.

We can achieve this by having an instance method on the type that returns an object that is almost the same, but has some modified fields:

```

public Deposit roll() {
    // Log audit event for rolling the date
    return new Deposit(amount, date.plusDays(1), payee);
}

public Deposit amend(double newAmount) {
    // Log audit event for amending the amount
    return new Deposit(newAmount, date, payee);
}

```

One problem that immutable objects potentially have is that they may need many parameters to be passed in to the factory method. This isn't always very convenient, especially when you may need to accumulate state from several sources before creating a new immutable object.

To solve this, we can use the Builder pattern. This is a combination of two constructs: a static inner class that implements a generic builder interface, and a private constructor for the immutable class itself.

The static inner class is the builder for the immutable class, and it provides the only way that a developer can get hold of new instances of the immutable type. One very common implementation is for the `Builder` class to have exactly the same fields as the immutable class, but to allow mutation of the fields.

This listing shows how you might use this to model a more complex view of a deposit.

Listing 5.3 Listing 4.3 Immutable objects and builders

```
public static class DepositBuilder implements Builder<Deposit> {
    private double amount;
    private LocalDate date;
    private Account payee;

    public DepositBuilder amount(double amount) {
        this.amount = amount;
        return this;
    }

    public DepositBuilder date(LocalDate date) {
        this.date = date;
        return this;
    }

    public DepositBuilder payee(Account payee) {
        this.payee = payee;
        return this;
    }

    @Override
    public Deposit build() {
        return new Deposit(amount, date, payee);
    }
}
```

The `Builder` is a generic top-level interface, which is usually defined like this:

```
public interface Builder<T> {
    T build();
}
```

We should note a couple of things about the builder. First of all, it is a so-called "SAM type" (for "Single Abstract Method") and it could, technically speaking, be used as the target type for a lambda expression. However, the purpose of the builder is to produce immutable instances - it is about gathering up state, not representing a function or callback. This means that although the builder *could* be used as a functional interface, in practice it will never be useful to do so.

For this reason, we do *not* decorate the interface with the `@FunctionalInterface` annotation. This is another good example of "just because you can do something, doesn't mean that you should".

Secondly, we should also notice that the builder is not thread-safe. The design implicitly assumes that the user knows not to share builders between threads. Instead, correct usage of the Builder API is for one thread to use a `Builder` to aggregate all needed state, and then produce an immutable object that *can* be trivially shared with other threads.

NOTE

If you find yourself wanting to share a builder between threads, take a moment to stop and reconsider your design and whether your domain needs refactoring.

Immutability is a very common pattern (not just in Java, but also in other languages, especially functional languages) and is one that has wide applicability.

One last point about immutable objects—the `final` keyword only applies to the object directly pointed to. As you can see in figure 5.7, the reference to the main object can't be assigned to point at object 3, but within the object, the reference to 1 can be updated to point at object 2. Another way of saying this is that a `final` reference can point at an object that has non-final fields. This is sometimes known as *shallow immutability*.

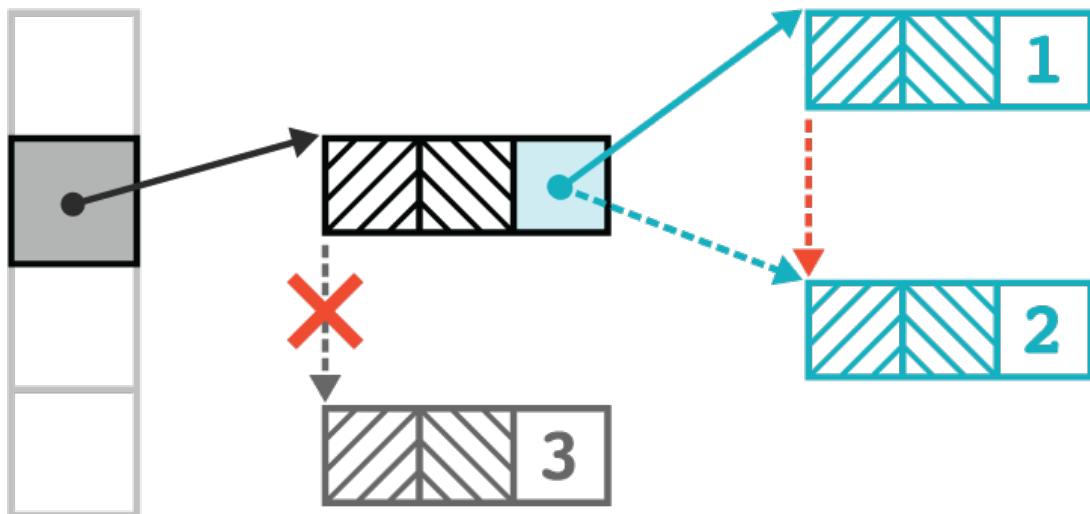


Figure 5.7 Immutability of value versus reference

Another way of seeing this is that it is perfectly possible to write:

```
final var numbers = new LinkedList<Integer>();
```

In this statement, the reference numbers and the integer objects contained in the list are immutable. However, the list object itself is still mutable because integer objects can still be added, removed, and replaced with the list.

Immutability is a very powerful technique, and you should use it whenever feasible. However, sometimes it's just not possible to develop efficiently with only immutable objects, because every change to an object's state requires a new object to be spun up. So we're sometimes left with the necessity of dealing with mutable objects.

In the next section, we'll discuss the often-misunderstood details of the Java Memory Model (JMM). Many Java programmers are aware of the JMM and have been coding to their own understanding of it without ever being formally introduced to it. If that sounds like you, this new understanding will build upon your informal awareness and place it onto firm foundations. The JMM is quite an advanced topic, so you can skip it if you're in a hurry to get on to the next chapter.

5.4 The Java Memory Model (JMM)

The JMM is described in section 17.4 of the Java Language Specification (JLS). This is quite a formal part of the spec, and it describes the JMM in terms of synchronization actions and some quite mathematical concepts - e.g. a *partial order* for operations.

This is great from the point of view of language theorists and implementers of the Java spec (compiler and VM makers), but it's worse for application developers who need to understand the details of how their multithreaded code will execute.

Rather than repeat the formal details, we'll list the most important rules here in terms of a couple of basic concepts: the *Synchronizes-With* and *Happens-Before* relationships between blocks of code.

- *Happens-Before*—This relationship indicates that one block of code fully completes before the other can start.
- *Synchronizes-With*—This means that an action will synchronize its view of an object with main memory before continuing.

If you've studied formal approaches to OO programming, you may have heard the expressions *Has-A* and *Is-A* used to describe the building blocks of object orientation. Some developers find it useful to think of *Happens-Before* and *Synchronizes-With* as being similar, basic conceptual building blocks - but for understanding Java concurrency rather than OO. However, we should emphasize that there is no direct technical connection between the two sets of concepts.

In figure 6.8 you can see an example of a volatile write that *Synchronizes-With* a later read access (for the `println()`).

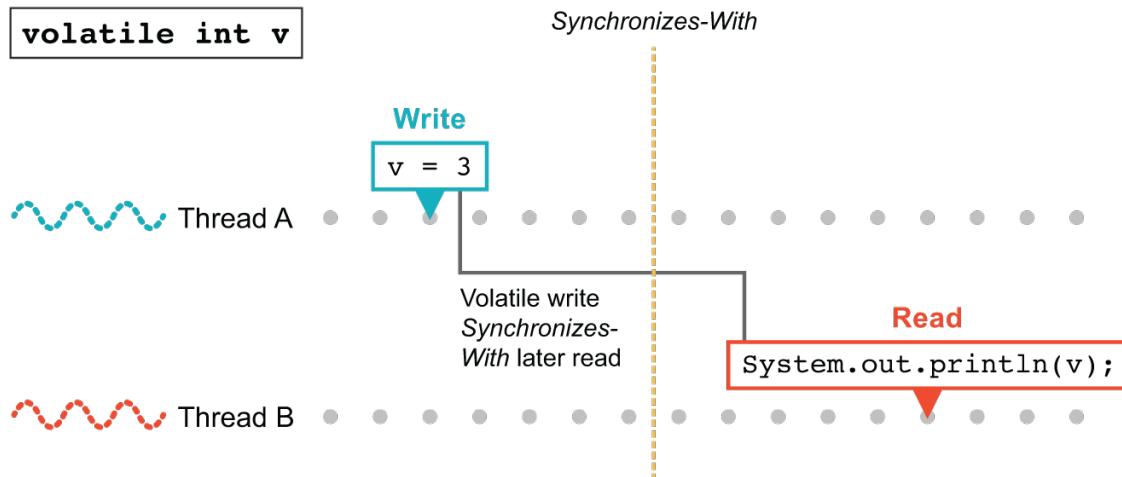


Figure 5.8 A Synchronizes-With example

The JMM has these main rules:

- An unlock operation on a monitor *Synchronizes-With* later lock operations.
- A write to a volatile variable *Synchronizes-With* later reads of the variable.
- If an action A *Synchronizes-With* action B, then A *Happens-Before* B.
- If A comes before B in program order, within a thread, then A *Happens-Before* B.

The general statement of the first two rules is that "releases happen before acquires." In other words, the locks that a thread holds when writing are released before the locks can be acquired by other operations (including reads).

For example, the rules guarantee that if one thread writes a value to a volatile variable, then any thread that later reads that variable will see the value that was written (assuming no other writes have taken place).

There are additional rules, which are really about sensible behavior:

- The completion of a constructor *Happens-Before* the finalizer for that object starts to run (an object has to be fully constructed before it can be finalized).
- An action that starts a thread *Synchronizes-With* the first action of the new thread.
- `Thread.join()` *Synchronizes-With* the last (and all other) actions in the thread being joined.
- If X *Happens-Before* Y and Y *Happens-Before* Z then X *Happens-Before* Z (transitivity).

These simple rules define the whole of the platform's view of how memory and synchronization works. Figure 5.9 illustrates the transitivity rule.

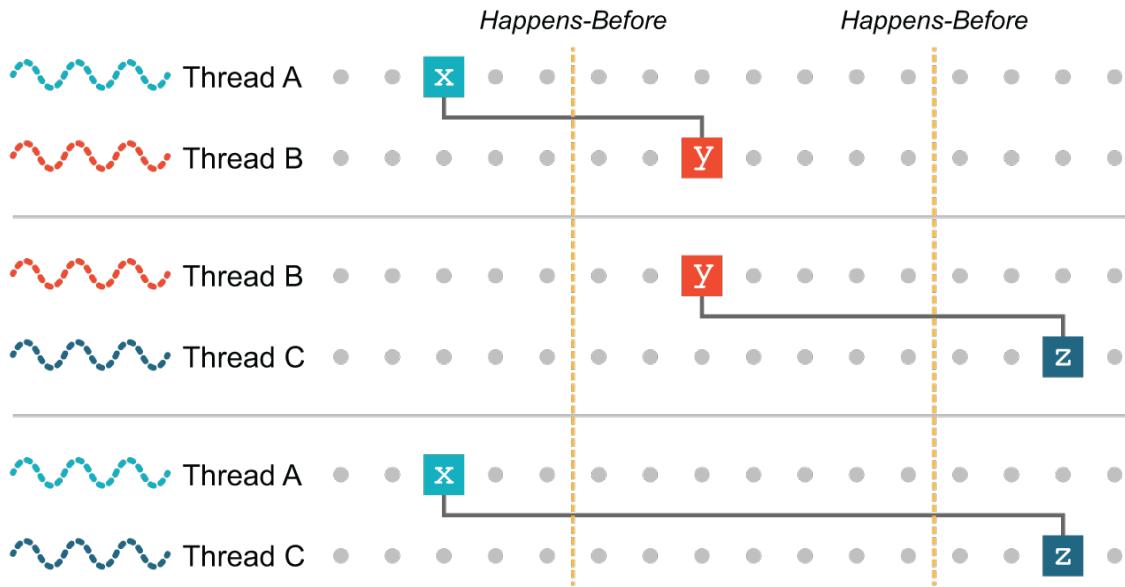


Figure 5.9 Transitivity of Happens-Before

NOTE

In practice, these rules are the minimum guarantees made by the JMM. Real JVMs may behave much better in practice than these guarantees suggest. This can be quite a pitfall for the developer because it's easy for the false sense of safety given by the behavior of a particular JVM to turn out to be just a quirk hiding an underlying concurrency bug.

From these minimum guarantees, it's easy to see why immutability is an important concept in concurrent Java programming. If objects can't be changed, there are no issues related to ensuring that changes are visible to all threads.

5.5 Understanding concurrency through bytecode

Let's discuss concurrency through the lens of a classic example - a bank account. Let's assume that a customer's account looks like this and that withdrawals and deposits are possible by calling methods.

We have provided both synchronized and unsynchronized implementations of the key methods.

```

public class Account {
    private double balance;

    public Account(int openingBalance) {
        balance = openingBalance;
    }

    public boolean rawWithdraw(int amount) {
        // Check to see amount > 0, throw if not
        if (balance >= amount) {
            balance = balance - amount;
            return true;
        }
        return false;
    }

    public void rawDeposit(int amount) {
        // Check to see amount > 0, throw if not
        balance = balance + amount;
    }

    public double getRawBalance() {
        return balance;
    }

    public boolean safeWithdraw(final int amount) {
        // Check to see amount > 0, throw if not
        synchronized (this) {
            if (balance >= amount) {
                balance = balance - amount;
                return true;
            }
        }
        return false;
    }

    public void safeDeposit(final int amount) {
        // Check to see amount > 0, throw if not
        synchronized (this) {
            balance = balance + amount;
        }
    }

    public double getSafeBalance() {
        synchronized (this) {
            return balance;
        }
    }
}

```

This set of methods will allow us to explore many common concurrency problems in Java.

NOTE	There is a reason why we are using the block form of synchronization at this stage, rather than the <code>synchronized</code> method modifier - we will explain why a bit later in the chapter.
-------------	--

We might also suppose that, if required, the class has 2-argument helper methods that look like this:

```
public boolean withdraw(int amount, boolean safe) {
    if (safe) {
        return safeWithdraw(amount);
    } else {
        return rawWithdraw(amount);
    }
}
```

Let's start by meeting one of the fundamental problems that multithreaded systems display that requires us to introduce some sort of protection mechanism.

5.5.1 Lost update

To demonstrate this common problem (or *antipattern*), known as *Lost Update*, let's look at the bytecode for the `rawDeposit()` method.

```
public void rawDeposit(int);
Code:
  0: aload_0
  1: aload_0
  2: getfield      #2  // Field balance:D      ①
  5: iload_1
  6: i2d
  7: dadd          ②
  8: putfield      #2  // Field balance:D      ③
 11: return
```

- ① Read balance from the object
- ② Add the deposit amount
- ③ Write new balance to the object

Let's introduce two threads of execution, called A and B. We can then imagine two deposits being attempted on the same account at once. By prefixing the instruction with the thread label, we can see individual bytecode instructions executing on different threads, but which are both affecting the same object.

NOTE

Remember that some bytecode instructions have parameters that follow them in the stream, which causes occasional "skips" in the instruction numbering.

Lost Update is the issue that it is possible, due to the non-deterministic scheduling of application threads, to end up with a bytecode sequence of reads and writes like this:

```

A0: aload_0
A1: aload_0
A2: getfield      #2  // Field balance:D      ①
A5: iload_1
A6: i2d
A7: dadd

// ....          Context switch A -> B

B0: aload_0
B1: aload_0
B2: getfield      #2  // Field balance:D      ②
B5: iload_1
B6: i2d
B7: dadd
B8: putfield      #2  // Field balance:D      ③
B11: return

// ....          Context switch B -> A

A8: putfield      #2  // Field balance:D      ④
A11: return

```

- ① Thread A reads a value from balance
- ② Thread B reads the same value from balance as A did
- ③ Thread B writes a new value back to the balance
- ④ Thread A overwrites the balance - B's update is lost

The updated balance is calculated by each thread by using the evaluation stack. The `dadd` opcode is the point where the updated balance is placed on the stack - but recall that every method invocation has its own, private evaluation stack.

This means that at the point `B7` in the above flow there are *two* copies of the updated balance - one in A's evaluation stack and one in B's. The two `putfield` operations at `B8` and `A8` then execute, but `A8` overwrites the value placed at `B8`.

This leads to the situation where both deposits appear to succeed, but only one of them actually shows up.

The account balance will register a deposit, but the code will still cause money to vanish from the account, because the balance field is read twice (with `getfield`), then written and overwritten (by the two `putfield` operations).

For example, in some code like this:

```

Account acc = new Account(0);
Thread tA = new Thread(() -> acc.rawDeposit(70));
Thread tB = new Thread(() -> acc.rawDeposit(50));
tA.start();
tB.start();
tA.join();
tB.join();

System.out.println(acc.getRawBalance());

```

it is possible for the final balance to be either 50 or 70 - but with both threads "successfully" depositing money - meaning that the code has paid in 120 but has lost some of it - a classic example of incorrect multithreaded code.

Be careful with the simple form of the code shown here. The full range of nondeterministic possibilities may not show up in such a simple example. Do not be fooled by this - when this code is combined into a large program the incorrectness will assuredly appear. Assuming that your code is OK because it's "too simple" or trying to cheat the concurrency model will inevitably end badly.

NOTE

There is an example (`AtmLoop`) that shows this effect in the source code repository - but it relies upon using a class we haven't met yet (`AtomicInteger`) - so we won't show it in full here. So, if you need to be convinced, please go and examine how the example behaves.

In general, access patterns like:

```

A: getfield
B: getfield
B: putfield
A: putfield

```

or

```

A: getfield
B: getfield
A: putfield
B: putfield

```

will cause problems for our account objects.

Recall that the operating system effectively causes non-deterministic scheduling of threads, so this type of interleaving is always possible - and Java objects live in the heap, so the threads are operating on shared, mutable data.

What we really need is to introduce a mechanism to somehow prevent this, and ensure that the ordering is always of the form:

```

...
A: getfield
A: putfield
...
B: getfield
B: putfield
...

```

This mechanism is synchronization - and it is our next subject.

5.5.2 Synchronization in bytecode

In Chapter 4, we introduced JVM bytecodes and briefly met `monitorenter` and `monitorexit`. A synchronized block is turned into these opcodes (we'll talk about synchronized methods a bit later). Let's see them in action by looking at an example we saw earlier (we're reproducing the Java code so it's close at hand):

```

public boolean safeWithdraw(final int amount) {
    // Check to see amount > 0, throw if not
    synchronized (this) {
        if (balance >= amount) {
            balance = balance - amount;
            return true;
        }
    }
    return false;
}

```

This is turned into 40 bytes of JVM bytecode:

```

public boolean safeWithdraw(int);
Code:
 0:  aload_0
 1:  dup
 2:  astore_2
 3:  monitorenter
 4:  aload_0
 5:  getfield      #2  // Field balance:D
 8:  iload_1
 9:  i2d
10:  dcmpl
11:  iflt       29
14:  aload_0
15:  aload_0
16:  getfield      #2  // Field balance:D
19:  iload_1
20:  i2d
21:  dsub
22:  putfield      #2  // Field balance:D
25:  iconst_1
26:  aload_2
27:  monitorexit
28:  ireturn
29:  aload_2
30:  monitorexit
31:  goto       39
34:  astore_3
35:  aload_2
36:  monitorexit
37:  aload_3
38:  athrow
39:  iconst_0
40:  ireturn

```

①

②

③

④

⑤

④

④

⑤

- ① Start of the synchronized block
- ② The if statement that checks the balance
- ③ Write the new value to the balance field
- ④ End of the synchronized block
- ⑤ Return from the method

There are a couple of oddities that the eagle-eyed reader might spot in the bytecode. First of all, let's look at the code paths. If the balance check succeeds then bytecodes 0-28 are executed with no jumps. If it fails, bytecodes 0-11 then a jump to 29-31 then a jump to 39-40 are executed.

This means that at first glance, there is no set of circumstances that will lead to bytecodes 34-38 being executed. This seeming discrepancy is actually explained by exception handling - some of the bytecode instructions (including `monitorenter`) can throw, and so there needs to be an exception handling code path.

The second puzzler is the return type of the method. In the Java code, it is declared as `boolean`, but we can see that the return instructions are `ireturn` - which is the integer variant of the return opcode.

In fact, there are no variant forms of instructions for bytes, shorts, chars, or booleans. This is

because these types are replaced by ints during the compilation process. This is a form of *type erasure* - which is one of the misunderstood aspects of Java's type system (especially as it is usually applied to the case of generics and type parameters).

Overall, the bytecode sequence above is more complex than the non-synchronized case, but should be possible to follow - we load the object to be locked onto the evaluation stack and then execute a `monitorenter` to acquire the lock. Let's assume the lock attempt succeeds.

Now, if any other thread then attempts to execute a `monitorenter` on the same object then the thread will block, and the second `monitorenter` instruction will not complete until the thread holding the lock executes a `monitorexit` and releases the lock.

This is how we deal with Lost Update - the `monitor` instructions are enforcing the ordering:

```
...
A: monitorenter
A: getfield
A: putfield
A: monitorexit
...
B: monitorenter
B: getfield
B: putfield
B: monitorexit
...
```

This provides the Happens-Before relationship between synchronized blocks: The end of one synchronized block Happens-Before the start of any other synchronized block on the same object - and this is guaranteed by the JMM.

We should also note that the Java source compiler ensures that every code path through a method that contains a `monitorenter` will result in a `monitorexit` being executed before the method terminates.

NOTE

The classfile verifier will in fact reject any class that tries to circumvent this rule, and exit from a method with a lock still held.

We can now see the basis for the claim that "synchronization is a co-operative mechanism in Java". Let's look at what happens when thread A calls `safeWithdraw()` and thread B calls `rawDeposit()`.

```

public boolean safeWithdraw(final int amount) {
    // Check to see amount > 0, throw if not
    synchronized (this) {
        if (balance >= amount) {
            balance = balance - amount;
            return true;
        }
    }
    return false;
}

```

We've reproduced the Java code once again for easy comparison.

```

public boolean safeWithdraw(int);
Code:
 0:  aload_0
 1:  dup
 2:  astore_2
 3:  monitorenter
 4:  aload_0
 5:  getfield      #2  // Field balance:D
 8:  iload_1
 9:  i2d
10: dcmpl
11: iflt           29
14:  aload_0
15:  aload_0
16:  getfield      #2  // Field balance:D
19:  iload_1
20:  i2d
21:  dsub
22:  putfield      #2  // Field balance:D
25:  iconst_1
26:  aload_2
27:  monitorexit
28:  ireturn

```

The depositing code is very simple - just one field read, an arithmetic operation, and a write back to the same field:

```

public void rawDeposit(int amount) {
    // Check to see amount > 0, throw if not
    balance = balance + amount;
}

```

The bytecode looks more complicated but actually isn't:

```

public void rawDeposit(int);
Code:
 0:  aload_0
 1:  aload_0
 2:  getfield      #2  // Field balance:D
 5:  iload_1
 6:  i2d
 7:  iadd
 8:  putfield      #2  // Field balance:D
11:  return

```

NOTE **The code for `rawDeposit()` does not contain any monitor instructions - and without a `monitorenter` the lock will never be checked.**

This means that an ordering like this, between two threads A and B is entirely possible:

```
// ...
A3: monitorenter
// ...

A14: aload_0
A15: aload_0
A16: getfield      #2  // Field balance:D

// ... Context switch A -> B

B0:  aload_0
B1:  aload_0
B2:  getfield      #2  // Field balance:D
B5:  iload_1
B6:  i2d
B7:  dadd
B8:  putfield      #2  // Field balance:D  ①

// ... Context switch B -> A

B11: return
A19: iload_1
A20: i2d
A21: dsub
A22: putfield      #2  // Field balance:D  ②
A25: iconst_1
A26: aload_2
A27: monitorexit
A28: ireturn
```

- ① Write to the balance (via unsynchronized method)
- ② Second write to the balance (via synchronized)

This is just our old friend Lost Update - but now occurs when one of the methods uses synchronization and one doesn't. The amount deposited has been lost - good news for the bank, but not so good for the customer.

The inescapable conclusion is this: To get the protections provided by synchronization, **all** methods must use it correctly.

5.5.3 Synchronized methods

Until now, we have been talking about the case of synchronized blocks, but what about the case of synchronized *methods*? We might guess that the compiler will insert synthetic `monitor` bytecodes, but that's actually not the case, as we can see if we change our safe methods to look like this:

```

public synchronized boolean safeWithdraw(final int amount) {
    // Check to see amount > 0, throw if not
    if (balance >= amount) {
        balance = balance - amount;
        return true;
    }
    return false;
}

// and the others...

```

Instead of showing up in the bytecode sequence, the `synchronized` modifier for the method actually shows up in the method's flags, as `ACC_SYNCHRONIZED`. We can see this by recompiling the method and notice that the `monitor` instructions have disappeared:

```

public synchronized boolean safeWithdraw(int);
Code:
  0: aload_0
  1: getfield      #2  // Field balance:D
  4: iload_1
  5: i2d
  6: dcmpl
  7: iflt         23
 10: aload_0
// ... no monitor instructions

```

When executing an `invoke` instruction, one of the first things the bytecode interpreter checks is to see if the method is `synchronized`. If it is, then the interpreter proceeds down a different code path - by first trying to acquire the appropriate lock. If the method does not have the `ACC_SYNCHRONIZED` then no such check is done.

This means that, just as we might expect, an unsynchronized method can execute at the same time as a synchronized one - as only one of them performs a check for the lock.

5.5.4 Unsynchronized Reads

A very common beginners error with Java concurrency is to assume that "only methods that write data need to be synchronized, reads are safe". This is emphatically not true, as we will demonstrate.

This false sense of security for reads sometimes occurs because the code example being reasoned about is a bit too simple.

What happens when we introduce a small ATM fee to our example - say 1% of the amount being withdrawn?

```

private final double atmFeePercent = 0.01;

public boolean safeWithdraw(final int amount, final boolean withFee) {
    // Check to see amount > 0, throw if not
    synchronized (this) {
        if (balance >= amount) {
            balance = balance - amount;
            if (withFee) {
                balance = balance - amount * atmFeePercent;
            }
            return true;
        }
    }
    return false;
}

```

The bytecode for this method is now a bit more complex:

```

public boolean safeWithdraw(int, boolean);
Code:
 0:  aload_0
 1:  dup
 2:  astore_3
 3:  monitorenter
 4:  aload_0
 5:  getfield      #2  // Field balance:D
 8:  iload_1
 9:  i2d
10:  dcmpl
11:  iflt       49          ①
14:  aload_0
15:  aload_0
16:  getfield      #2  // Field balance:D
19:  iload_1
20:  i2d
21:  dsub
22:  putfield      #2  // Field balance:D          ②
25:  iload_2
26:  ifeq       45
29:  aload_0
30:  aload_0
31:  getfield      #2  // Field balance:D
34:  iload_1
35:  i2d
36:  aload_0
37:  getfield      #5  // Field atmFeePercent:D
40:  dmul
41:  dsub
42:  putfield      #2  // Field balance:D          ③
45:  iconst_1
46:  aload_3
47:  monitorexit
48:  ireturn
49:  aload_3
50:  monitorexit
51:  goto       61
54:  astore       4
56:  aload_3
57:  monitorexit
58:  aload       4
60:  athrow
61:  iconst_0
62:  ireturn

```

- ① Compare balance to amount
- ② Account balance updated
- ③ Fee applied and balance updated again

Note that there are now two `putfield` instructions, as `safeWithdraw()` takes a boolean parameter that determines whether or not a fee should be charged. The fact that there are two separate updates is what raises the potential for a concurrency bug.

The code for reading the raw balance is very simple:

```
public double getRawBalance();
Code:
 0:  aload_0
 1:  getfield      #2  // Field balance:D
 4:  dreturn
```

However, this can be interleaved with the withdraw-with-fee code like this:

```
A14:  aload_0
A15:  aload_0
A16:  getfield      #2  // Field balance:D
A19:  iload_1
A20:  i2d
A21:  dsub
A22:  putfield      #2  // Field balance:D  ①
A25:  iload_2
A26:  ifeq        45
A29:  aload_0
A30:  aload_0
A31:  getfield      #2  // Field balance:D

// ... Context switch A -> B

B0:  aload_0
B1:  getfield      #2  // Field balance:D  ②
B4:  dreturn

// ... Context switch B -> A

A34:  iload_1
A35:  i2d
A36:  aload_0
A37:  getfield      #5  // Field atmFeePercent:D
A40:  dmul
A41:  dsub
A42:  putfield      #2  // Field balance:D
```

- ① Balance written with amount (but not fee) deducted
- ② Balance read while full withdraw is still being processed

With an unsynchronized read, there is the possibility of a *phantom read* - a value that does not actually correspond to a real state of the system. If you're familiar with SQL databases, then this may remind you of performing a read partway through a database transaction.

NOTE

You might be tempted to think "I know the bytecodes", and optimize your code based on that. You should resist this temptation for several reasons - for example, what happens when you have handed over your code and it is maintained by other developers who do not understand the context or consequences of seemingly-harmless code changes?

The conclusion: There is no get-out clause for "just reads". If even one code path fails to use synchronization correctly, then the resulting code is not thread-safe and so is incorrect in a multi-threaded environment.

Let's move on and take a look at how deadlock shows up in bytecode.

5.5.5 Deadlock revisited

Suppose the bank wants to add the ability to transfer money between accounts into our code. An initial version of this code might look like this:

```
public boolean naiveSafeTransferTo(Account other, int amount) {
    // Check to see amount > 0, throw if not
    synchronized (this) {
        if (balance >= amount) {
            balance = balance - amount;
            synchronized (other) {
                other.rawDeposit(amount);
            }
            return true;
        }
    }
    return false;
}
```

This produces quite a long bytecode listing, so we have shortened it by omitting the by-now-familiar sequence for checking that the balance can support the withdrawal and some synthetic exception-handling blocks.

NOTE

There are now two account objects, and each of them has a lock. To be safe, we need to coordinate access to both locks - the lock belonging to `this` and that belonging to `other`.

We will need to deal with **two** pairs of `monitor` instructions - with each pair dealing with the lock of a different object:

```

public boolean naiveSafeTransferTo(Account, int);
Code:
 0: aload_0
 1: dup
 2: astore_3
 3: monitorenter          ①

// Omit the usual balance checking bytecode

14: aload_0
15: aload_0
16: getfield      #2  // Field balance:D
19: iload_2
20: i2d
21: dsub
22: putfield      #2  // Field balance:D
25: aload_1
26: dup
27: astore        4
29: monitorenter          ②
30: aload_1
31: iload_2
32: invokevirtual #6  // Method rawDeposit:(I)V
35: aload        4
37: monitorexit          ③
38: goto         49

// Omit exception handling code

49: iconst_1
50: aload_3
51: monitorexit          ④
52: ireturn

// Omit exception handling code

```

- ① Acquire lock on this object
- ② Acquire lock on other object
- ③ Release lock on other object
- ④ Release lock on this object

Imagine two threads that are trying to transfer money between the same two accounts - let's call the threads A and B. Let's further suppose that the threads are executing transactions that are labelled by the sending account. So thread A is trying to send money from object A to object B and vice versa.

```

A0: aload_0
A1: dup
A2: astore_3
A3: monitorenter      ①

// Omit the usual balance checking bytecode

B0: aload_0
B1: dup
B2: astore_3
B3: monitorenter      ②

// Omit the usual balance checking bytecode

B14: aload_0
B15: aload_0
B16: getfield      #2  // Field balance:D
B19: iload_2
B20: i2d
B21: dsub
B22: putfield      #2  // Field balance:D
B25: aload_1
B26: dup
B27: astore        4
B29: ...           ③

A14: aload_0
A15: aload_0
A16: getfield      #2  // Field balance:D
A19: iload_2
A20: i2d
A21: dsub
A22: putfield      #2  // Field balance:D
A25: aload_1
A26: dup
A27: astore        4
A29: ...           ④

```

- ① Lock acquired on account object A (by thread A)
- ② Lock acquired on account object B (by thread B)
- ③ Thread B tries to acquire lock on object A. Fails & blocks
- ④ Thread A tries to acquire lock on object B. Fails & blocks

After executing this sequence, neither thread can make progress. Worse yet, only thread A can release the lock on object A and only thread B can release the lock on object B, so these two threads are permanently blocked by the synchronization mechanism and these method calls will never complete.

By viewing the deadlock anti-pattern at a bytecode level, we can see clearly what actually causes it.

5.5.6 Deadlock resolved, revisited

To solve this problem, as we discussed earlier, we need to ensure that the locks are always acquired in the same order by every thread. One way to do this is by creating an ordering on the threads - say by introducing a unique account number and implementing the rule: "acquire the lock corresponding to the lowest account id first".

NOTE

For objects that don't have numeric ids, we will need to do something different, but the general principle of using an unambiguous total order still applies.

This produces a bit more complexity - and to do it completely correctly we need a guarantee that the account ids are not reused. We can do this by introducing a `static int` field - which holds the next account id to be allocated, and only update it in a `synchronized` method, like this:

```
private static int nextAccountId = 1;

private final int accountId;

private static synchronized int getAndIncrementNextAccountId() {
    int result = nextAccountId;
    nextAccountId = nextAccountId + 1;
    return result;
}

public Account(int openingBalance) {
    balance = openingBalance;
    atmFeePercent = 0.01;
    accountId = getAndIncrementNextAccountId();
}

public int getAccountId() {
    return accountId;
}
```

We don't need to synchronize the `getAccountId()` method because the field is `final` and can't change.

```

public boolean safeTransferTo(final Account other, final int amount) {
    // Check to see amount > 0, throw if not
    if (accountId == other.getAccountId()) {
        // Can't transfer to your own account
        return false;
    }

    if (accountId < other.getAccountId()) {
        synchronized (this) {
            if (balance >= amount) {
                balance = balance - amount;
                synchronized (other) {
                    other.rawDeposit(amount);
                }
                return true;
            }
        }
        return false;
    } else {
        synchronized (other) {
            synchronized (this) {
                if (balance >= amount) {
                    balance = balance - amount;
                    other.rawDeposit(amount);
                    return true;
                }
            }
        }
        return false;
    }
}

```

The resulting Java code is, of course, a little asymmetrical.

NOTE

By avoiding holding any locks for any longer than necessary makes it clear which parts of the code actually require locks.

The above produces a very long bytecode listing, but let's break it down by parts. First off, we check the ordering of the account IDs:

```

// Elide balance and account equality checks
13: aload_0
14: getfield      #8   // Field accountId:I
17: aload_1
18: invokevirtual #10  // Method getAccountId():I
21: if_icmpge     91

```

This means that if `A < B` (which it is), then we step on to instruction 24, otherwise we jump ahead to 91.

```

24: aload_0
25: dup
26: astore_3
27: monitorenter
28: aload_0
29: getfield      #3   // Field balance:D
32: iload_2
33: i2d
34: dcmpl
35: iflt          77

```

①

②

- ① Start of synchronized (this) {
- ② If insufficient funds, bail out to offset 77 (further on)

Let's follow the branch where the sending account has sufficient funds to continue, so control falls through to bytecode 38, which is the start of the `balance = balance - amount;` statement in the Java code:

```

38: aload_0
39: aload_0
40: getfield      #3    // Field balance:D
43: iload_2
44: i2d
45: dsub
46: putfield      #3    // Field balance:D
49: aload_1
50: dup
51: astore         4
53: monitorenter          ①
54: aload_1
55: iload_2
56: invokevirtual #9    // Method rawDeposit:(I)V
59: aload           4
61: monitorexit          ②
62: goto            73
// Omit exception handling code
73: iconst_1
74: aload_3
75: monitorexit          ③
76: ireturn

```

- ① Start of synchronized (other) {
- ② End of synchronized (other) {
- ③ End of synchronized (this) {

For completeness, let's show the code path used in the case of insufficient balance in the sending account - we basically just unlock the monitor on `this` and return.

```

77: aload_3
78: monitorexit          ①
79: goto            89
// Omit exception handling code
89: iconst_0
90: ireturn

```

- ① End of synchronized (this) {

Note that some of the instructions (such as `invoke` and `monitor` instructions) may throw exceptions, so we are as usual ignoring the bytecode handlers for those exceptions.

The rest of the method looks like this:

```
91:  aload_1  
// ...  
// Highly similar, but for the other branch
```

Let's look at what happens with two threads, remembering that the account id of A < B.

There is one additional complication that we have now - because the local variables (used in instructions such as `aload_0`) are different between the two threads. To draw out this distinction, we'll slightly mangle the bytecode by labelling the local variable with the thread as well, so we'll write `aload_A0` and `aload_A1` for clarity.

```

A24: aload_A0
A25: dup
A26: astore_A3
A27: monitorenter ①

// Elide balance check

A38: aload_A0
A39: aload_A0
A40: getfield #3 // Field balance:D

// .... Context switch A -> B

B91: aload_B1
B92: dup
B93: astore_B3
B94: monitorenter ②

// .... Context switch B -> A

A43: iload_A2
A44: i2d
A45: dsub
A46: putfield #3 // Field balance:D
A49: aload_A1
A50: dup
A51: astore A4
A53: monitorenter ③
A54: aload_A1
A55: iload_A2
A56: invokevirtual #9 // Method rawDeposit:(I)V
A59: aload A4
A61: monitorexit ④
A62: goto 73

// Omit exception handling code

A73: iconst_A1
A74: aload_A3
A75: monitorexit ⑤

// .... Context switch A -> B

B95: aload_B0
B96: dup
B97: astore B4
B99: monitorenter
// ...
B132: ireturn

// .... Context switch B -> A

A76: ireturn

```

- ① Lock acquired on object A by thread A
- ② Lock attempted on object A by thread B - blocks
- ③ Lock acquired on object B by thread A
- ④ Lock released on object B by thread A
- ⑤ Lock released on object A by thread A - thread B can resume

This is, without doubt, a complex listing.

The key insight is that `A0 == B1` so that locking these two objects will always induce a blocking call in the second thread. The invariant `A < B` ensures that thread B is sent down the alternate branch.

5.5.7 Volatile access

What does `volatile` look like in bytecode? Let's take a look at an important pattern - *Volatile Shutdown* - to help answer this.

The Volatile Shutdown pattern helps solve the problem of inter-thread communication that we touched upon earlier when we met the dangerous and deprecated `stop()` method earlier in the chapter.

Consider a simple class that is responsible for doing some work. In the simplest case, we will assume that work comes in discrete units, with a well-defined "complete" status for each unit:

```
public class TaskManager implements Runnable {
    private volatile boolean shutdown = false;

    public void shutdown() {
        shutdown = true;
    }

    @Override
    public void run() {
        while (!shutdown) {
            // do some work - e.g. process a work unit
        }
    }
}
```

The intent of the pattern is hopefully clear. All the time the `shutdown` flag is false, work units will continue to be processed. If it ever flips to true, then the `TaskManager` will, after it has completed its current work unit, exit the `while` loop and the thread will exit cleanly, in a "graceful shutdown".

The more subtle point is derived from the Java Memory Model: Any write to a volatile variable *Happens-Before* all subsequent reads of that variable.

This means that as soon as another thread calls `shutdown()` on the `TaskManager` object, then the flag is changed to `true` and the effect of that change is guaranteed to be visible on the next read of the flag - before the next work unit is accepted.

The Volatile Shutdown pattern produces bytecode like this:

```

public class TaskManager implements java.lang.Runnable {
    private volatile boolean shutdown;

    public TaskManager();
    Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
        4: aload_0
        5: iconst_0
        6: putfield      #2                  // Field shutdown:Z
        9: return

    public void shutdown();
    Code:
        0: aload_0
        1: iconst_1
        2: putfield      #2                  // Field shutdown:Z
        5: return

    public void run();
    Code:
        0: aload_0
        1: getfield      #2                  // Field shutdown:Z
        4: ifne          10
        7: goto          0
        10: return
}

```

If you look carefully, you can see that the volatile nature of `shutdown` does not appear anywhere except on the field definition. There are no additional clues on the opcodes - and it is accessed using the standard `getfield` and `putfield` opcodes.

NOTE

This is because `volatile` is a hardware access mode - and produces a CPU instruction that says to ignore the cache hardware and instead read or write directly from main memory.

The only difference is in how `putfield` and `getfield` behave - the implementation of the bytecode interpreter will have separate code paths for volatile and standard fields.

In fact, *any* piece of physical memory can be accessed in a volatile manner - and as we will see later on - this is not the only access mode that is possible. The volatile case is merely a common case of access semantics that James Gosling and the original designers of Java chose to encode in the core of the language, by making it a keyword that can apply to fields.

5.6 Summary

Concurrency is one of the most important features of the Java platform, and a good developer will increasingly need a solid understanding of it. We've reviewed the underpinnings of Java's concurrency and the design forces that occur in multithreaded systems. We've discussed the Java Memory Model and low-level details of how the platform implements concurrency.

This chapter isn't intended to be a complete statement of everything you'll ever need to know

about concurrency — it's enough to get you started and give you an appreciation of what you'll need to learn more about, and to stop you being dangerous when writing concurrent code. But you'll need to know more than we can cover here if you're going to be a truly first-rate developer of multithreaded code. There are a number of excellent books about nothing but Java concurrency — one of the best is *Java Concurrency in Practice* by Brian Goetz and others (Addison-Wesley Professional, 2006).

- Java's threads are a low-level abstraction
- Multithreading is present even in the Java bytecode
- The Java Memory Model is very flexible, but makes minimal guarantees
- Synchronization is a cooperative mechanism - all threads must participate to achieve safety
- Never use `Thread.stop()`

In the next chapter, we're going to go up the stack, and introduce the concurrency library known as `java.util.concurrent` and show how the principles and concepts we've introduced here show up when building practical multithreaded applications.



JDK concurrency libraries

This chapter covers

- Atomic Classes
- Locks Classes
- Concurrent data structures
- Blocking Queues
- Futures and `CompletableFuture`
- Executors

In this chapter we'll cover what every well-grounded developer should know about `java.util.concurrent` and how to use the toolbox of concurrency building blocks it provides. The aim is that by the end of the chapter, you'll be ready to start applying these libraries and concurrency techniques in your own code.

6.1 Building blocks for modern concurrent applications

As we saw last chapter, Java has supported concurrency since the very beginning. However, with the advent of Java 5 (which was itself over 15 years ago), a new way of thinking about concurrency in Java emerged. This was spearheaded by the package `java.util.concurrent`, which contained a rich new toolbox for working with multithreaded code.

NOTE

This toolbox has been enhanced with subsequent versions of Java, but the classes and packages that were introduced with Java 5 still work the same way and they're still very valuable to the working developer.

If you (still!) have existing multithreaded code that is still based solely on the older (pre-Java 5) approaches, you should consider refactoring it to use `java.util.concurrent`. In our

experience, your code will be improved if you make a conscious effort to port it to the newer APIs — the greater clarity and reliability will be well worth the effort expended to migrate in almost all cases.

We're going to take a tour through some of the headline classes in `java.util.concurrent` and related packages, such as the atomic and locks packages. We'll get you started using the classes and look at examples of use cases for them.

You should also read the Javadoc for them and try to build up your familiarity with the packages as a whole — most developers find that the higher level of abstraction that they provide makes concurrent programming much easier.

6.2 Atomic classes

The package `java.util.concurrent.atomic` contains several classes that have names starting with `Atomic`. For example, `AtomicBoolean`, `AtomicInteger`, `AtomicLong` and `AtomicReference`. These classes are one of the simplest examples of a *concurrency primitive* - a class that can be used to build workable, safe concurrent applications.

CAUTION **Atomic classes don't inherit from the similarly named classes, so `AtomicBoolean` can't be used in place of a `Boolean`, and an `AtomicInteger` isn't an `Integer` (but it does extend `Number`).**

The point of an atomic is to provide thread-safe mutable variables. Each of the four classes provides access to a single variable of the appropriate type.

NOTE **The implementations of the atomics are written to take advantage of modern processor features, so they can be non-blocking (lock-free) if support is available from the hardware and OS - which it will be for virtually all modern systems.**

The access provided is lock-free on almost all modern hardware and so the atomics behave in a similar way to a volatile field. However, they are wrapped in a class API that goes further than what's possible with volatiles. This API includes atomic (meaning all-or-nothing) methods for suitable operations - including state-dependent updates (which are impossible to do with volatile variables without using a lock). The end result is that atomics can be a very simple way for a developer to avoid race conditions on shared data.

NOTE **If you're curious as to how atomics are implemented, we will discuss the details in Chapter 16, when we talk about internals and the class `sun.misc.Unsafe`.**

A common use case for atomics is to implement something similar to sequence numbers, as you might find provided by an SQL database. This capability is accessed by using methods such as the atomic `getAndIncrement()` on the `AtomicInteger` or `AtomicLong` classes.

Let's look at how we would rewrite the `Account` example from Chapter 5 to use an atomic:

```
private static AtomicInteger nextAccountId = new AtomicInteger(1);

private final int accountId;
private double balance;

public Account(int openingBalance) {
    balance = openingBalance;
    accountId = nextAccountId.getAndIncrement();
}
```

As each object is created, we make a call to `getAndIncrement()` on the static instance of `AtomicInteger`, which returns us an `int` value, and atomically increments the mutable variable. This atomicity guarantees that it is impossible for two objects to share the same `accountId` - which is exactly the property that we want (just like a database sequence number).

NOTE

We could add the `final` qualifier to the atomic, but it's not necessary as the field is `static` and the class doesn't provide any way to mutate the field.

For another example, here is how we would rewrite our volatile shutdown example to use an `AtomicBoolean`:

```
public class TaskManager implements Runnable {
    private final AtomicBoolean shutdown = new AtomicBoolean(false);

    public void shutdown() {
        shutdown.set(true);
    }

    @Override
    public void run() {
        while (!shutdown.get()) {
            // do some work - e.g. process a work unit
        }
    }
}
```

As well as these examples, the `AtomicReference` is also used to implement atomic changes - but to objects. The general pattern is that some modified (possibly immutable) state is built optimistically, and can then be "swapped in" by using a *Compare-And-Swap* (CAS) operation on an `AtomicReference`.

Next, let's examine how `java.util.concurrent` models the core of the classic synchronization approach — the `Lock` interface.

6.3 Lock classes

The block-structured approach to synchronization is based around a simple notion of what a lock is. This approach has a number of shortcomings:

- There is only one type of lock.
- It applies equally to all synchronized operations on the locked object.
- The lock is acquired at the start of the synchronized block or method.
- The lock is released at the end of the block or method.
- The lock is either acquired or the thread blocks — no other outcomes are possible.

If we were going to re-engineer the support for locks, there are several things we could potentially change for the better:

- Add different types of locks (such as reader / writer locks).
- Not restrict locks to blocks (allow a lock in one method and unlock in another).
- If a thread cannot acquire a lock (for example, if another thread has the lock), allow the thread to back out or carry on or do something else — a `tryLock()`.
- Allow a thread to attempt to acquire a lock and give up after a certain amount of time.

The key to realizing all of these possibilities is the `Lock` interface in `java.util.concurrent.locks`. This interface ships with a couple of implementations:

- `ReentrantLock` — This is essentially the equivalent of the familiar lock used in Java synchronized blocks, but more flexible.
- `ReentrantReadWriteLock` — This can provide better performance in cases where there are many readers but few writers.

NOTE

There are other implementations, both within the JDK and written by third parties - but these are by far the most common.

The `Lock` interface can be used to completely replicate any functionality that is offered by block-structured concurrency. For example, here is the example from Chapter 5 for how to avoid deadlock rewritten to use `ReentrantLock`.

We need to add a lock object as a field to the class, because we will no longer be relying on the intrinsic lock on the object.

We also need to maintain the principle that locks are always acquired in the same order. In our example the simple protocol we maintain is that the lock on the object with the lowest account id is acquired *first*.

Listing 6.1 Rewriting deadlock example to use ReentrantLock

```

private final Lock lock = new ReentrantLock();

public boolean transferTo(SafeAccount other, int amount) {
    // We also need code to check to see amount > 0, throw if not
    // ...

    if (accountId == other.getAccountId()) {
        // Can't transfer to your own account
        return false;
    }

    if (accountId < other.getAccountId()) {                                ①
        lock.lock();
        try {
            if (balance >= amount) {
                balance = balance - amount;
                other.lock.lock();                                         ①
                try {
                    other.deposit(amount);
                } finally {
                    other.lock.unlock();
                }
                return true;
            }
        } finally {
            lock.unlock();
        }
        return false;
    } else {                                                               ②
        other.lock.lock();                                                 ②
        try {
            lock.lock();
            try {
                if (balance >= amount) {
                    balance = balance - amount;
                    other.deposit(amount);
                    return true;
                }
            } finally {
                lock.unlock();
            }
        } finally {
            other.lock.unlock();
        }
        return false;
    }
}

```

- ① On this branch the current object has a lower account id
- ② On this branch the other object has a lower account id

The pattern of an initial call to `lock()` combined with a `try ... finally` block, where the lock is released in the `finally` is a great addition to your toolbox.

NOTE

The locks, like much of `java.util.concurrent`, **relies on a class called** `AbstractQueuedSynchronizer` **to implement their functionality.**

The pattern works very well if you're replicating a situation that is similar to one where you'd have used block-structured concurrency. On the other hand, if you need to pass around the `Lock` objects (such as by returning it from a method), you can't use this pattern.

6.3.1 Condition objects

Another aspect of the API provided by `java.util.concurrent` are the *condition objects*. These are objects that play the same role in the API as `wait()` and `notify()` do in the original intrinsic API, but are more flexible. They provide the ability for threads to wait indefinitely for some condition, and to be woken up when that condition becomes true.

However, unlike the intrinsic API (where the object monitor only has a single condition for signalling), the `Lock` interface allows the programmer to create as many condition objects as they like. This allows a separation of concerns - for example the lock can have multiple, disjoint groups of methods that can use separate conditions.

A condition object (which implements the interface `Condition`) is created by calling the `newCondition()` method on a lock object (one that implements the `Lock` interface).

As well as condition objects, the API provides a number of *latches* and *barriers* as concurrency primitives that may be useful in some circumstances.

6.4 CountDownLatch

The `CountDownLatch` is a simple concurrency primitive that provides a *consensus barrier* - it allows for multiple threads to reach a co-ordination point and wait until the barrier is released.

This is achieved by providing an int value (the `count`) when constructing a new instance of `CountDownLatch`. After that point, two methods are used to control the latch: `countDown()` and `await()`.

The former reduces the count by 1, and the latter causes the calling thread to block until the count reaches 0 (it does nothing if the count is already 0 or less).

In the following listing, the latch is used by each `Runnable` to indicate when it has completed its assigned work.

Listing 6.2 Using latches to signal between threads

```

public static class Counter implements Runnable {
    private final CountDownLatch latch;
    private final int value;
    private final AtomicInteger count;

    public Counter(CountDownLatch l, int v, AtomicInteger c) {
        this.latch = l;
        this.value = v;
        this.count = c;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException __) {
        }
        count.addAndGet(value); ①
        latch.countDown(); ②
    }
}

```

- ① Update the count value atomically
- ② Decrement the latch

Note that the `countDown()` method is non-blocking, so once the latch has been decremented, then the thread running the Counter code will exit.

We also need some driver code:

```

var latch = new CountDownLatch(2);
var count = new AtomicInteger();
for (int i = 0; i < 2; i = i + 1) {
    var r = new Counter(latch, i, count);
    new Thread(r).start();
}

try {
    latch.await();
    System.out.println("Total: " + count.get());
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

In the code, the latch is set up with a quorum value (in this case 2). Next, the same number of threads are created and initialized, so that processing can begin. The main thread awaits the latch and blocks until it is released. Each worker thread will perform a sleep and then `countDown()` once it has finished. This means that the main thread will not proceed until both of the threads have completed their processing. This situation is show in Figure 6.1

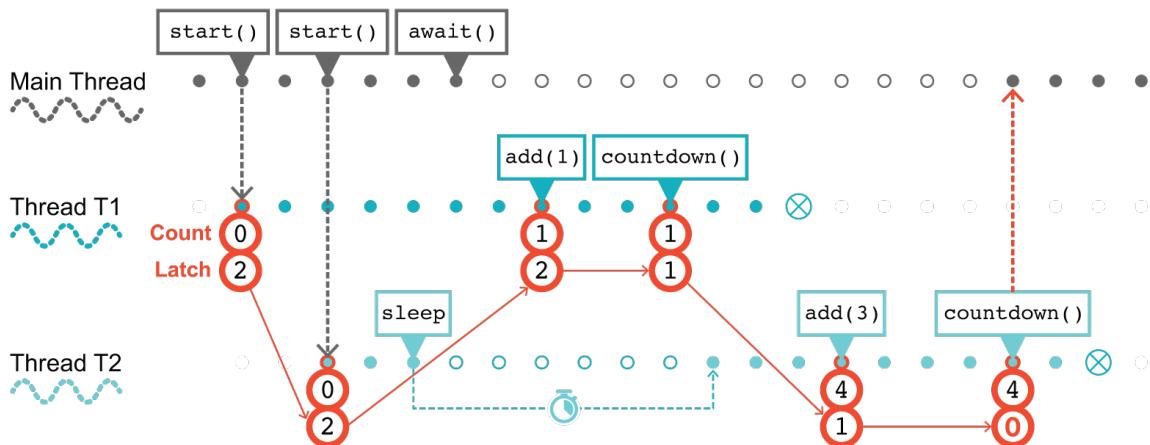


Figure 6.1 Using a CountDownLatch

To provide another example of a good use case for `CountDownLatch`, consider an application that needs to pre-populate several caches with reference data before the server is ready to receive incoming requests. We can easily achieve this by using a shared latch, a reference to which is held by each cache population thread.

When each cache finishes loading, the `Runnable` populating it counts down the latch and exits. When all the caches are loaded, the main thread (which has been awaiting the latch opening) can proceed and is ready to mark the service as up and begin handling requests.

The next class we'll discuss is one of the most useful classes in the multithreaded developer's toolkit: the `ConcurrentHashMap`.

6.5 ConcurrentHashMap

The `ConcurrentHashMap` class provides a concurrent version of the standard `HashMap`. In general, maps are a very useful (& common) data structure for building concurrent applications. This is due, at least in part, to the shape of the underlying data structure. Let's take a closer look at the basic `HashMap` to understand why.

6.5.1 Understanding a simplified HashMap

As you can see from figure 6.2, the classic Java `HashMap` uses a function (the *hash function*) to determine which *bucket* it will store the key-value pair in. This is where the "hash" part of the class's name comes from.

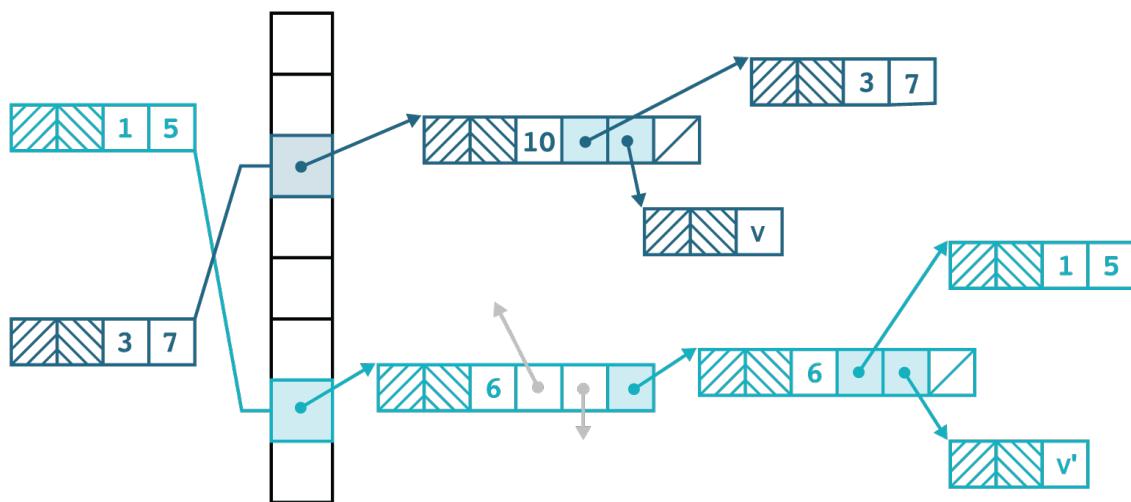


Figure 6.2 The classic view of a HashMap

The key-value pair is actually stored in a linked list (known as the *hash chain*) that starts from the bucket corresponding to the index obtained by hashing the key.

In the Github project that accompanies this book is a simplified implementation of a `Map<String, String>` - the `Dictionary` class. This class is actually based on the form of the `HashMap` that shipped as part of Java 7.

NOTE

Modern Javas ship a `HashMap` implementation that is significantly more complex, so in this explanation we focus on a simpler version where the design concepts are more clearly visible.

The basic class has only two fields - the main data structure, and the `size` field which caches the size of the map for performance reasons:

```
public class Dictionary implements Map<String, String> {
    private Node[] table = new Node[8];
    private int size;

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }
}
```

These rely on a helper class, called a `Node`, which represents a key-value pair and implements the interface `Map.Entry`:

```

static class Node implements Map.Entry<String, String> {
    final int hash;
    final String key;
    String value;
    Node next;

    Node(int hash, String key, String value, Node next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    public final String getKey() { return key; }
    public final String getValue() { return value; }
    public final String toString() { return key + "=" + value; }

    public final int hashCode() {
        return Objects.hashCode(key) ^ Objects.hashCode(value);
    }

    public final String setValue(String newValue) {
        String oldValue = value;
        value = newValue;
        return oldValue;
    }

    public final boolean equals(Object o) {
        if (o == this)
            return true;
        if (o instanceof Node) {
            Node e = (Node)o;
            if (Objects.equals(key, e.getKey()) &&
                Objects.equals(value, e.getValue()))
                return true;
        }
        return false;
    }
}

```

To look for a value in the map, we use the `get()` method, which relies on a couple of helper methods, `hash()` and `indexFor()`

```

@Override
public String get(Object key) {
    if (key == null)
        return null;
    int hash = hash(key);
    for (Node e = table[indexFor(hash, table.length)];
         e != null;
         e = e.next) {
        Object k = e.key;
        if (e.hash == hash && (k == key || key.equals(k)))
            return e.value;
    }
    return null;
}

static final int hash(Object key) {
    int h = key.hashCode();
    return h ^ (h >>> 16); ①
}

static int indexFor(int h, int length) {
    return h & (length - 1); ②
}

```

- ① Bitwise operation to make sure that the hash value is positive
- ② Bitwise operation to make sure that the index is within the size of the table

The `get()` method first of all deals with the irritating null case. Following that, we use the key object's hash code to construct an index into the array `table`. There is an unwritten assumption that the size of `table` is a power of 2, which means that the operation of `indexFor()` is basically a modulo operation, which ensures that the return value is a valid index into `table`.

NOTE

This is a classic example of a situation where a human mind can determine that an exception (in this case `ArrayIndexOutOfBoundsException`) will never be thrown, but the compiler cannot.

Now that we have an index into `table`, we use it to select the relevant hash chain for our lookup operation. We start at the head, and walk down the hash chain. At each step we evaluate whether we've found our key object:

```
if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
    return e.value;
```

If we have, then we return the corresponding value. This is why we store keys and values as pairs (really as `Node` instances) - to allow for this approach.

The `put()` method is somewhat similar to the above:

```
@Override
public String put(String key, String value) {
    if (key == null)
        return null;

    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Node e = table[i]; e != null; e = e.next) {
        Object k = e.key;
        if (e.hash == hash && (k == key || key.equals(k))) {
            String oldValue = e.value;
            e.value = value;
            return oldValue;
        }
    }

    Node e = table[i];
    table[i] = new Node(hash, key, value, e);

    return null;
}
```

This version of a hashed data structure is not 100% production quality, but it is intended to demonstrate the basic behaviour and approach to the problem, so that the concurrent case can be understood.

6.5.2 Limitations of Dictionary

Before we proceed to the concurrent case, we should mention that some methods from `Map` are not supported by our toy implementation, `Dictionary`. Specifically, `putAll()`, `keySet()`, `values()`, or `entrySet()` (which need to be defined, because the class implements `Map`) will simply throw new `UnsupportedOperationException()`.

We do not support these methods purely and solely due to complexity. As we will see several times in the book - the Java Collections interfaces are large and feature-rich. This is good for the end user, because they have a lot of power - but it means that there are a lot more methods that an implementor must supply.

In particular, methods like `keySet()` require an implementation of `Map` to supply instances of `Set` - and this frequently results in needing to write an entire implementation of the `Set` interface as an inner class. That is too much extra complexity for our examples, so we just don't support those methods in our toy implementation.

NOTE

As we will see later in the book, the monolithic, complex, imperative design of the Collections interfaces presents various problems when we start to think about functional programming in detail.

The simple dictionary class works well, within its limitations. However, it does not guard against two scenarios:

- The need to resize `table` as the number of elements stored increases
- Defending against keys that implement a pathological form of `hashCode()`

The first of these is a serious limitation. A major point of a hashed data structure is to reduce the expected complexity operations down from $O(N)$ to $O(\log N)$ for e.g. value retrieval. If the table is not resized as the number of elements held in the map increases then this complexity gain is lost. A real implementation would have to deal with the need to resize the table as the map grows.

6.5.3 Approaches to a concurrent Dictionary

As it stands, `Dictionary` is obviously not thread-safe.

Consider two threads - one trying to delete a certain key and the other trying to update the value associated with it. Depending on the ordering of operations, it is entirely possible for both the deletion and the update to report that they succeeded when in fact only one of them did.

To resolve this, there are two fairly obvious (if naive) ways to making `Dictionary` (and, by extension, general Java `Map` implementations) concurrent.

First off, is the fully-synchronization approach, which we met in Chapter 5.

The punchline is not hard to predict - this approach is unfeasible for most practical systems due to performance overhead. However, it's worth a small diversion to look at how we might implement it.

There are two easy ways to achieve simple thread safety here. The first is to copy the `Dictionary` class - let's call it `ThreadSafeDictionary` and then make all of its methods synchronized. This works, but involves a lot of duplicated, cut-and-paste code.

Alternatively, we can use a synchronized wrapper to provide *delegation* - aka forwarding - to an underlying object that actually houses the dictionary. Here's how we can do that.

```
public final class SynchronizedDictionary extends Dictionary {
    private final Dictionary d;

    private SynchronizedDictionary(Dictionary delegate) {
        d = delegate;
    }

    public static SynchronizedDictionary of(Dictionary delegate) {
        return new SynchronizedDictionary(delegate);
    }

    @Override
    public synchronized int size() {
        return d.size();
    }

    @Override
    public synchronized boolean isEmpty() {
        return d.isEmpty();
    }

    // ... other methods
}
```

There are numerous problems with this - the most important of which is that the object `d` already exists, and is not synchronized. This is setting ourselves up to fail - other code may modify `d` outside of a synchronized block or method and we find ourselves in exactly the situation we discussed in the last chapter.

This is not the right approach for concurrent data structures.

We should mention that, in fact, the JDK provides just such an implementation - the `synchronizedMap()` method provided in the `Collections` class. It works about as well, and is about as widely used, as you might expect.

A second approach is to appeal to immutability. As we will say, and say again, the Java Collections are large, and complex interfaces. One way in which this manifests is that the

assumption of mutability is baked throughout the collections - in no sense is it a separable concern that some implementations may choose, or not, to express - all implementations of `Map` and `List` must implement the mutating methods.

Due to this, it might seem as though there is no way to model a data structure in Java that is both immutable and conforms to the Java Collections APIs - if it conforms to the APIs, the class must also provide an implementation of the mutation method.

However, there is a deeply unsatisfactory back door. An implementation of an interface can always throw `UnsupportedOperationException` if it has not implemented a certain method. From a language design point of view, this is of course terrible. An interface contract should be exactly that - a contract.

Unfortunately, this mechanism and convention pre-dates Java 8 (and the arrival of default methods) and thus represents an attempt to encode a difference between a "mandatory" method and an "optional" one - at a time when no such distinction actually existed in the Java language.

It is a bad mechanism and practice (especially since `UnsupportedOperationException` is a runtime exception), but we could use it something like this:

```
public final class ImmutableDictionary extends Dictionary {
    private final Dictionary d;

    private ImmutableDictionary(Dictionary delegate) {
        d = delegate;
    }

    public static ImmutableDictionary of(Dictionary delegate) {
        return new ImmutableDictionary(delegate);
    }

    @Override
    public int size() {
        return d.size();
    }

    @Override
    public String get(Object key) {
        return d.get(key);
    }

    @Override
    public String put(String key, String value) {
        throw new UnsupportedOperationException();
    }

    // other mutating methods also throw UnsupportedOperationException
}
```

It can be argued that this is something of a violation of object-oriented principles - the expectation from the user is that this is a valid implementation of `Map<String, String>` and yet, if a user tries to mutate an instance, then an unchecked exception is thrown. This can legitimately be seen as a safety hazard.

NOTE

This is basically the compromise that `Map.of()` has to make - it needs to fully implement the interface, and so has to resort to throwing on mutating method calls.

This is also not the only issue with this approach - another drawback is that this is, of course, subject to the same basic flaw as we saw for the synchronized case - a mutable object still exists and can be referenced (and mutated) via that route, violating the basic criteria that we were trying to achieve.

Let us draw a veil over these attempts, and try to look for something better.

6.5.4 Using ConcurrentHashMap

Having shown a simple map implementation, and discussed approaches that we could use to make it concurrent, it's time to meet the `ConcurrentHashMap`. In some ways, this is the easy part - it is an extremely easy to use class and in most cases is a drop-in replacement for `HashMap`.

The key point about the `ConcurrentHashMap` is that it is safe for multiple threads to update it at once. To see why we need this, let's see what happens when we have two threads adding entries to a map simultaneously:

```
var map = new HashMap<String, String>();
var SIZE = 10_000;

Runnable r1 = () -> {
    for (int i = 0; i < SIZE; i = i + 1) {
        map.put("t1" + i, "0");
    }
    System.out.println("Thread 1 done");
};

Runnable r2 = () -> {
    for (int i = 0; i < SIZE; i = i + 1) {
        map.put("t2" + i, "0");
    }
    System.out.println("Thread 2 done");
};

Thread t1 = new Thread(r1);
Thread t2 = new Thread(r2);
t1.start();
t2.start();
try {
    t1.join();
    t2.join();
} catch (InterruptedException __) {
    System.err.println("Threads interrupted");
}
System.out.println("Count: " + map.size());
```

If we run this code, we will see a different manifestation of our old friend, the Lost Update antipattern - the output value for count will be less than $2 * \text{SIZE}$. However, in the case of concurrent access to a map, the situation is actually much, much worse.

The most dangerous behavior of `HashMap` under concurrent modification does not always manifest at small sizes. However, if we increase the value of `SIZE` it will, eventually, manifest itself.

If we increase `SIZE` to, say, `1_000_000` then we are likely to see the behavior - one of the threads making updates to `map` will fail to finish. That's right - one of the threads can (and will) get stuck in an actual infinite loop. This makes `HashMap` totally unsafe for use in multithreaded applications (and the same is true of our example `Dictionary` class).

On the other hand, if we replace `HashMap` with `ConcurrentHashMap`, then we can see that the concurrent version behaves properly - no infinite loops and no instances of Lost Update. It also has the nice property that, no matter what you do to it, map operations will never throw a `ConcurrentModificationException`.

Let's take a very brief look at how this is achieved. It turns out that Figure 6.2, which shows the implementation of `Dictionary`, also points the way to a useful multithreaded generalization of `Map` that is much better than either of our two previous attempts. It is based on the following insight:

Instead of needing to lock the whole structure when making a change, it's only necessary to lock the hash chain (aka bucket) that's being altered or read.

We can see how this works in Figure 6.3 - the implementation has moved the lock down onto the individual hash chains. This technique is known as *lock striping*, and it enables multiple threads to access the map, provided they are operating on different chains.

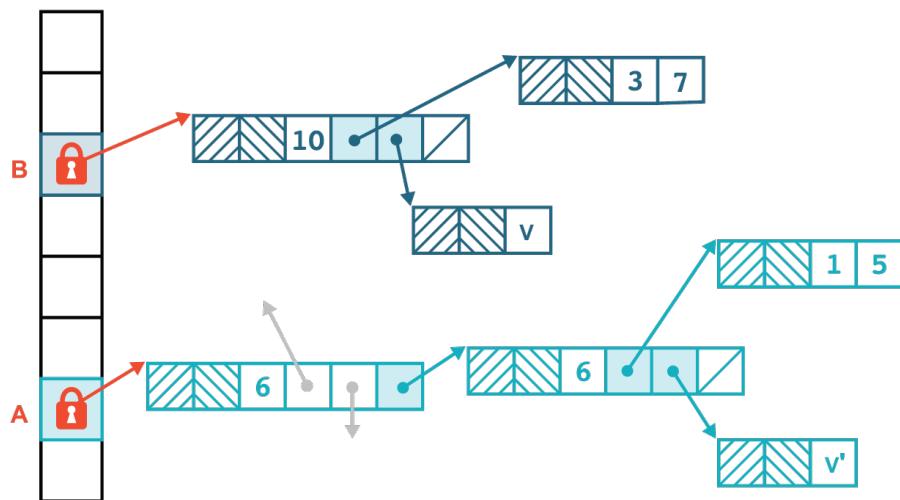


Figure 6.3 Lock striping

Of course, if two threads need to operate on the same chain, then they will still exclude each other, but in general, this provides better throughput than synchronizing the entire map.

NOTE

Recall that as the number of elements in the map increases, the table of buckets will resize - meaning that as more and more elements are added to a ConcurrentHashMap, it will become able to deal with more and more threads in an efficient manner.

The ConcurrentHashMap achieves this behavior, but there are some additional low-level details that most developers won't need to worry about too much. In fact, the implementation of ConcurrentHashMap changed substantially in Java 8 and it is now more complex than the design that we have described here.

Using ConcurrentHashMap can be almost be too simple - in many cases all that is needed is:

"If you have a multithreaded program and need to share data, then just use a Map, and have the implementation be a 'ConcurrentHashMap'"

In fact, if there is ever a chance that a Map might need to be modified by more than one thread, then you should always use the concurrent implementation. It does use considerably more resources than a plain HashMap and will have worse throughput due to the synchronization of some operations - but, as we'll discuss in Chapter 7 - those inconveniences are nothing when compared to the possibility of a race condition leading to Lost Update or an infinite loop.

Finally, we should also note that ConcurrentHashMap actually implements the ConcurrentMap interface, which extends Map. It originally contained some new methods to provide thread-safe modifications:

- `putIfAbsent()` - Adds the key/value pair to the HashMap if the key isn't already present.
- `remove()` - Safely removes the key/value pair if the key is present.
- `replace()` - The implementation provides two different forms of this method for safe replacement in the HashMap

However, with Java 8, some of these methods were retrofitted to the Map interface as default methods, for example:

```
default V putIfAbsent(K key, V value) {
    V v = get(key);
    if (v == null) {
        v = put(key, value);
    }
    return v;
}
```

This means that the gap between ConcurrentHashMap and Map has narrowed somewhat in recent versions of Java - but don't forget that despite this HashMap remains thread-unsafe. If you want to share data safely between threads, you should use ConcurrentHashMap.

Overall, the `ConcurrentHashMap` is one of the most useful classes in `java.util.concurrent`. It provides additional multithreaded safety and higher performance than synchronization, and it has no serious drawbacks in normal usage. The counterpart to it for `List` is the `CopyOnWriteArrayList`, which we'll discuss next.

6.6 CopyOnWriteArrayList

We can, of course, apply the two unsatisfactory concurrency patterns that we saw in the last section to `List` as well. Fully-synchronized and immutable (but with mutating methods that throw runtime exception) lists are as easy to write down as they are for maps - and they work no better than they do for maps.

Can we do better? Unfortunately, the linear nature of the list is not helpful here. Even in the case of a linked list, multiple threads attempting to modify the list raise the possibility of contention - for example in workloads that have a large percentage of append operations.

One alternative that does exist is the `CopyOnWriteArrayList` class. As the name suggests, this type is a replacement for the standard `ArrayList` class that has been made thread-safe by the addition of *copy-on-write semantics*. This means that any operations that mutate the list will create a new copy of the array backing the list (as shown in figure 6.4). This also means that any iterators created don't have to worry about any modifications that they didn't expect.

The iterators are guaranteed not to throw `ConcurrentModificationException` and will not reflect any additions, removals, or changes to the list since the iterator was created - except of course that (as usual in Java) the list elements can still mutate - it is only the list that cannot.

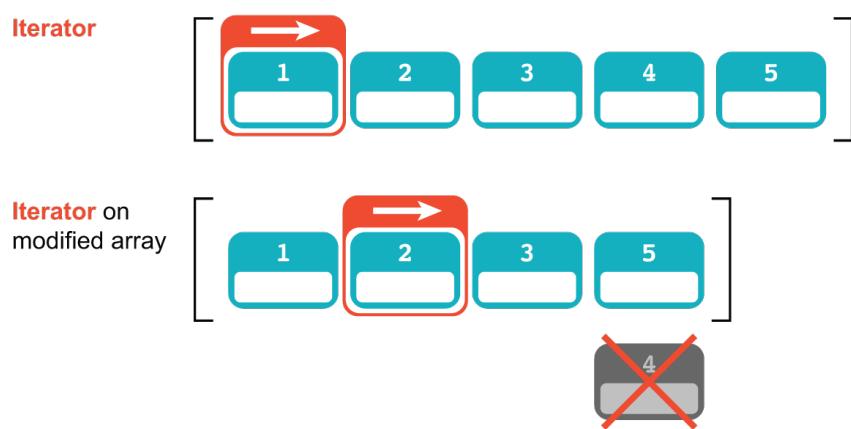


Figure 6.4 Copy-on-write array

This implementation is usually too expensive for general use, but may be a good option when traversal operations vastly outnumber mutations, and when the programmer does not want the headache of synchronization yet wants to rule out the possibility of threads interfering with each other.

Let's take a quick look at how the core idea is implemented. The key methods are `iterator()`, which always returns a new `COWIterator` object:

```
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}
```

and `add()`, `remove()` and other mutation methods. The mutation methods always replace the delegate array with a new, cloned and modified copy of the array. To protect the array, this must be done within a synchronized block, so the `CopyOnWriteArrayList` class has an internal lock that is just used as a monitor (and note the comment on it):

```
/**
 * The lock protecting all mutators. (We have a mild preference
 * for builtin monitors over ReentrantLock when either will do.)
 */
final transient Object lock = new Object();

private transient volatile Object[] array;
```

Then, operations such as `add()` can be protected:

```
public boolean add(E e) {
    synchronized (lock) {
        Object[] es = getArray();
        int len = es.length;
        es = Arrays.copyOf(es, len + 1);
        es[len] = e;
        setArray(es);
        return true;
    }
}
```

This makes the `CopyOnWriteArrayList` less efficient than the `ArrayList` for general operations, for several reasons:

- Synchronisation of mutation operations
- Volatile storage (i.e. array)
- `ArrayList` will only allocate memory when a resize of the underlying array is required, `CopyOnWriteArrayList` allocates and copies on every mutation.

Creating the iterator stores a reference to the array as it exists at that point in time. Further modifications to the list cause a new copy to be created, so the iterator will then be pointing at a past version of the array.

```

static final class COWIterator<E> implements ListIterator<E> {
    /** Snapshot of the array */
    private final Object[] snapshot;
    /** Index of element to be returned by subsequent call to next */
    private int cursor;

    COWIterator(Object[] es, int initialCursor) {
        cursor = initialCursor;
        snapshot = es;
    }
    ...
}

```

Note that `COWIterator` implements `ListIterator` and so, according to the interface contract, is required to support list mutation methods - but for simplicity's sake, the mutators all throw `UnsupportedOperationException`.

The approach taken by `CopyOnWriteArrayList` to shared data may be useful when a quick, consistent snapshot of data (which may occasionally be different between readers) is more important than perfect synchronization. This is seen reasonably often in scenarios that involve non-mission-critical data and the copy-on-write approach avoids the performance hit associated with synchronization.

Let's look at an example of copy-on-write in action.

Listing 6.3 Copy-on-write example

```

var ls = new CopyOnWriteArrayList(List.of(1, 2, 3));
var it = ls.iterator();
ls.add(4);
var modifiedIt = ls.iterator();
while (it.hasNext()) {
    System.out.println("Original: " + it.next());
}
while (modifiedIt.hasNext()) {
    System.out.println("Modified: " + modifiedIt.next());
}

```

This code is specifically designed to illustrate the behavior of an `Iterator` under copy-on-write semantics. It produces output like this:

```

Original: 1
Original: 2
Original: 3
Modified: 1
Modified: 2
Modified: 3
Modified: 4

```

In general, the use of the `CopyOnWriteArrayList` class does require a bit more thought than using `ConcurrentHashMap`, which is basically a drop-in concurrent replacement for `HashMap`. This is because of performance issues — the copy-on-write property means that if the list is altered, the entire array must be copied.

This means that if changes to the list are common, compared to read accesses, this approach won't necessarily yield high performance.

In general, the `CopyOnWriteArrayList` makes different tradeoffs than `synchronizedList()`. The latter synchronizes on all operations, so reads from different threads can block each other, which is not true for a COW data structure. On the other hand, `CopyOnWriteArrayList` copies the backing array on every mutation, whereas the synchronized version only does so when the backing array is full (same behavior as `ArrayList`).

However, as we'll say repeatedly in chapter 7, reasoning about code from first principles is extremely difficult - the only way to reliably get well-performing code is to test, retest, and measure the results.

Later on, in Chapter 15, we'll meet the concept of a *persistent data structure* - which is another way of approaching concurrent data handling. The Clojure programming language makes very heavy use of persistent data structures, and the `CopyOnWriteArrayList` (and `CopyOnWriteArraySet`) is one example implementation of them.

Let's move on. The next major common building block of concurrent code in `java.util.concurrent` is the `Queue`. This is used to hand off work elements between threads, and it is used as the basis for many flexible and reliable multithreaded designs.

6.7 Blocking queues

The queue is a wonderful abstraction for concurrent programming. The queue provides a simple and reliable way to distribute processing resources to work units (or to assign work units to processing resources, depending on how you want to look at it).

There are a number of patterns in multithreaded Java programming that rely heavily on the thread-safe implementations of `Queue`, so it's important that you fully understand it. The basic `Queue` interface is in `java.util`, because it can be an important pattern even in single-threaded programming, but we'll focus on the multithreaded use cases.

One very common use case, and the one we'll focus on, is the use of a queue to transfer work units between threads. This pattern is often ideally suited for the simplest concurrent extension of `Queue` — the `BlockingQueue`.

The `BlockingQueue` is a queue that has two additional special properties:

- When trying to `put()` to the queue, it will cause the putting thread to wait for space to become available if the queue is full.
- When trying to `take()` from the queue, it will cause the taking thread to block if the queue is empty.

These two properties are very useful because if one thread (or pool of threads) is outstripping the ability of the other to keep up, the faster thread is forced to wait, thus regulating the overall system. This is illustrated in figure 6.5.

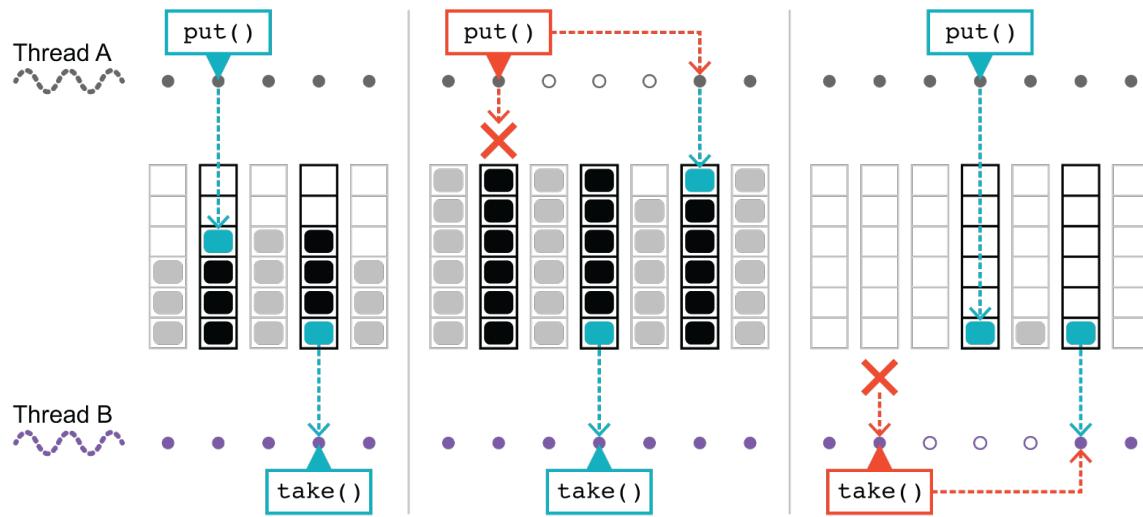


Figure 6.5 The BlockingQueue

Java ships with two basic implementations of the `BlockingQueue` interface: the `LinkedBlockingQueue` and the `ArrayBlockingQueue`. They offer slightly different properties; for example, the array implementation is very efficient when an exact bound is known for the size of the queue, whereas the linked implementation may be slightly faster under some circumstances.

However, the real difference between the implementations is in the implied semantics. Whilst the linked variant can be constructed with a size limit, it is usually created without one - which leads to an object with a queue size of `Integer.MAX_VALUE`. This is effectively infinite - a real application would never be able to recover from a backlog of over 2 billion items in one of its queues.

So, although in theory the `put()` method on `LinkedBlockingQueue` can block, in practice it never does. This means that the threads that are writing to the queue can effectively proceed at an unlimited rate.

In contrast, the `ArrayBlockingQueue` has a fixed size for the queue - the size of the array that backs it. This means that if the producer threads are putting objects into the queue faster than they are being processed by receivers then at some point the queue will fill completely, and further attempts to call `put()` will block and the producer threads will be forced to slow their rate of task production.

This property of the `ArrayBlockingQueue` is one form of what is known as *backpressure*, which

is an important aspect of engineering concurrent and distributed systems.

Let's see the `BlockingQueue` in action in an example — altering the account example to use queues and threads. The aim of the example will be to get rid of the need to lock both account objects. The basic architecture of the application is shown in figure 6.6.

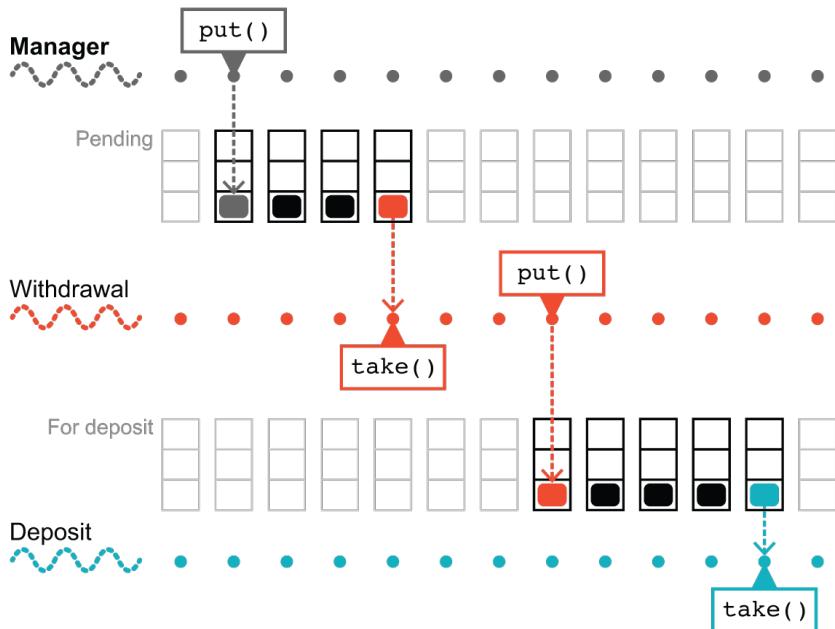


Figure 6.6 Handling accounts with queues

We start by introducing a `AccountManager` class with these fields:

Listing 6.4 The Account Manager

```
public class AccountManager {
    private ConcurrentHashMap<Integer, Account> accounts =
        new ConcurrentHashMap<>();
    private volatile boolean shutdown = false;

    private BlockingQueue<TransferTask> pending =
        new LinkedBlockingQueue<>();
    private BlockingQueue<TransferTask> forDeposit =
        new LinkedBlockingQueue<>();
    private BlockingQueue<TransferTask> failed =
        new LinkedBlockingQueue<>();

    private Thread withdrawals;
    private Thread deposits;
```

The blocking queues contain `TransferTask` objects, which are simple data carriers that denote the transfer to be made:

```

public class TransferTask {
    private final Account sender;
    private final Account receiver;
    private final int amount;

    public TransferTask(Account sender, Account receiver, int amount) {
        this.sender = sender;
        this.receiver = receiver;
        this.amount = amount;
    }

    public Account sender() {
        return sender;
    }

    public int amount() {
        return amount;
    }

    public Account receiver() {
        return receiver;
    }

    // Other methods elided
}

```

There is no additional semantics for the transfer - the class is just a dumb *data carrier* type.

NOTE

The `TransferTask` type is very simple and in Java 17 could be written as a record type (which we met in Chapter 3)

The `AccountManager` class provides functionality for accounts to be created, and for transfer tasks to be submitted:

```

public Account createAccount(int balance) {
    var out = new Account(balance);
    accounts.put(out.getAccountId(), out);
    return out;
}

public void submit(TransferTask transfer) {
    try {
        pending.put(transfer);
    } catch (InterruptedException e) {
        try {
            failed.put(transfer);
        } catch (InterruptedException x) {
            // Log at high criticality
        }
    }
}

```

The real work of the account manager is handled by the two threads that manage the transfer tasks between the queues. Let's look at the withdraw operation first:

```

public void init() {
    Runnable withdraw = () -> {
        while (!shutdown) {
            try {
                var task = pending.take();
                var sender = task.sender();
                if (sender.withdraw(task.amount())) {
                    forDeposit.put(task);
                } else {
                    failed.put(task);
                }
            } catch (InterruptedException e) {
                // Log at critical and proceed to next item
            }
        }
        // Drain pending queue to failed or log
    };
}

```

The deposit operation is somewhat similar:

```

Runnable deposit = () -> {
    LOOP:
    while (!shutdown) {
        TransferTask task;
        try {
            task = forDeposit.take();
        } catch (InterruptedException e) {
            // Log at critical and proceed to next item
            continue LOOP;
        }
        var receiver = task.receiver();
        receiver.deposit(task.amount());
    }
    // Drain forDeposit queue to failed or log
};

init(withdraw, deposit);
}

```

The package-private overload of the `init()` method is used to start the background threads. It exists as a separate method to allow for easier testing.

```

void init(Runnable withdraw, Runnable deposit) {
    withdrawals = new Thread(withdraw);
    deposits = new Thread(deposit);
    withdrawals.start();
    deposits.start();
}

```

We need some code to drive this:

```

var manager = new AccountManager();
manager.init();
var acc1 = manager.createAccount(1000);
var acc2 = manager.createAccount(20_000);

var transfer = new TransferTask(acc1, acc2, 100); ①
manager.submit(transfer);
try {
    Thread.sleep(5000); ②
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println(acc1);
System.out.println(acc2);
manager.shutdown();
manager.await();

```

- ① Submit transfer from acc1 to acc2
- ② Sleep to allow time for the transfer to execute

This produces output like this:

```

Account{accountId=1, balance=900.0,
    lock=java.util.concurrent.locks.ReentrantLock@58372a00[Unlocked]}
Account{accountId=2, balance=20100.0,
    lock=java.util.concurrent.locks.ReentrantLock@4dd8dc3[Unlocked]}

```

However, the code as written does not execute cleanly - despite the calls to `shutdown()` and `await()`. This is because of the blocking nature of the calls used. Let's look at Figure 6.7 to see why:

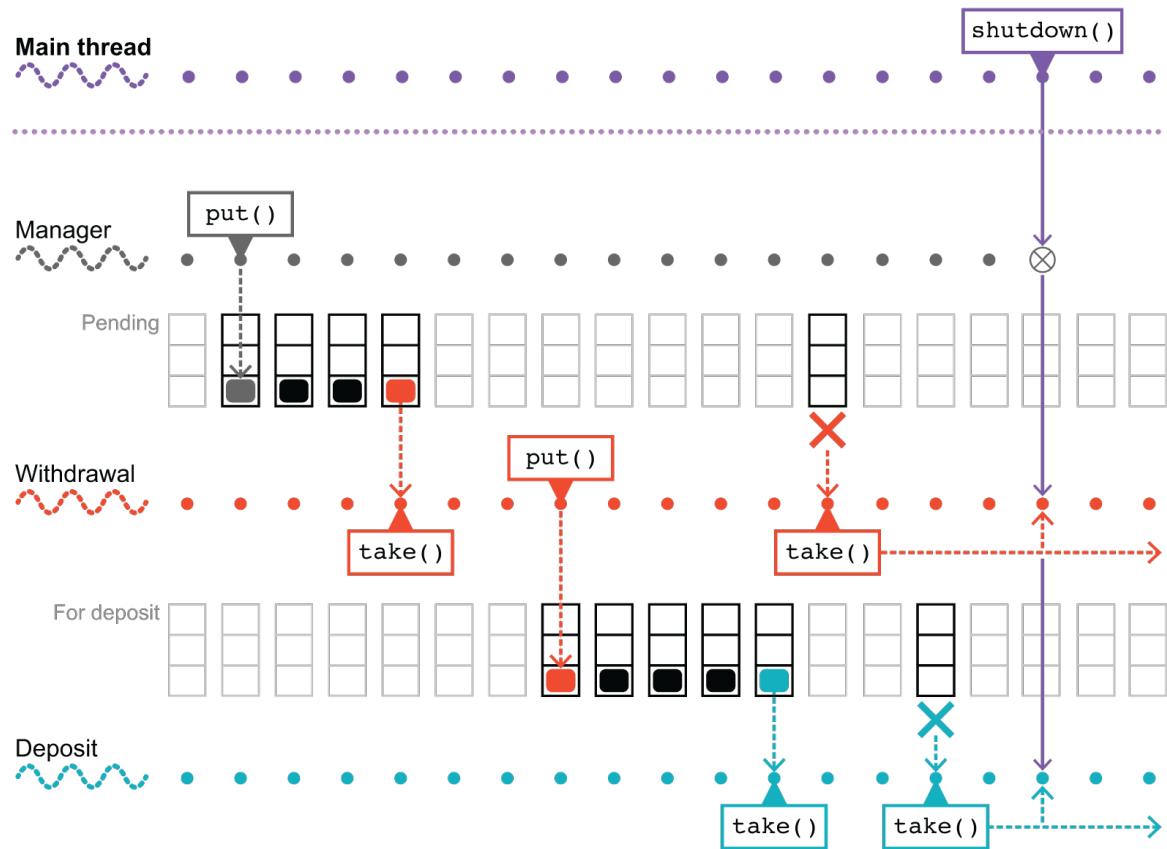


Figure 6.7 Incorrect shutdown sequence

When the main code calls `shutdown()` the volatile boolean flag is flipped to true, so every subsequent read of the boolean will see the value true.

Unfortunately, both the withdrawal and depositing threads are both blocked in calls to `take()` - because the queues are empty. If an object was somehow placed into the `Pending` queue then the withdrawal thread would process it, and then place the object into the `forDeposit` queue (assuming the withdrawal succeeds). The withdrawal thread would at this point exit the while loop and the thread would terminate normally.

In turn, the depositing thread will now see the object in the `forDeposit` queue and will wake up, take it, process it and then exit its own while loop and also terminate normally.

However, this clean termination process depends upon there still being tasks in the queue. In the edge case of an empty queue the threads will sit in their blocking `take()` calls forever.

To solve this issue, let's explore the full range of methods that are provided by the blocking queue implementations.

6.7.1 Using BlockingQueue APIs

The interface `BlockingQueue` actually provides three separate strategies for interacting with it. To understand the differences between the strategies consider the possible behaviors that an API could display in the following scenario:

A thread attempts to insert an item in a capacity-restricted queue that is currently unable to accommodate the item (i.e. the queue is full).

Logically, there are three possibilities. The insertion call could:

- Block until space in the queue frees up
- Return a value (perhaps boolean false) indicating failure
- Throw an exception

The same three possibilities would, of course, occur in the converse situation (attempting to take an item from an empty queue).

The first of these possibilities is realized by the `take()` and `put()` methods that we have already met.

NOTE

The second and third are the options provided by the `Queue` interface, which is the super interface of `BlockingQueue`.

The second option provides a non-blocking API that returns special values, and is manifested in the methods `offer()` and `poll()`. If insertion into the queue cannot be completed, then `offer()` fails fast and returns false. The programmer must examine the return code and take appropriate action.

Similarly, `poll()` immediately returns `null` on failure to retrieve from the queue. It might seem a bit odd to have a non-blocking API on a class explicitly named `BlockingQueue` but it is actually useful (and also required as a consequence of the inheritance relationship between `BlockingQueue` and `Queue`).

In fact, `BlockingQueue` provides an additional overload of the non-blocking methods. These methods provide the capability of polling or offering with a timeout, to allow the thread encountering issues to back out from its interaction with the queue and do something else instead.

We can modify the account manager in Listing 6.4 to use the non-blocking APIs with timeout, like so:

```

Runnable withdraw = () -> {
    LOOP:
    while (!shutdown) {
        try {
            var task = pending.poll(5, TimeUnit.SECONDS);      ①
            if (task == null) {
                continue LOOP;                                ②
            }
            var sender = task.sender();
            if (sender.withdraw(task.amount())) {
                forDeposit.put(task);
            } else {
                failed.put(task);
            }
        } catch (InterruptedException e) {
            // Log at =critical and proceed to next item
        }
    }
    // Drain pending queue to failed or log
};

```

- ① If the timer expires, `poll()` returns null
- ② Explicit use of a Java loop label to make it clear what is being continued

Similar modifications should be made for the deposit thread as well.

This solves the shutdown problem that we outlined in the previous subsection, as now the threads cannot block forever in the retrieval methods. Instead, if no object arrives before the timeout then the poll will still return and provide the value `null`. The test then continues the loop, but the visibility guarantees of the volatile boolean ensure that the while loop condition is now met and the loop is exited and the thread shuts down cleanly.

This means that overall, once the `shutdown()` method has been called, the account manager will shut down in bounded time, which is the behavior we want.

NOTE	Additional code and checks will still be required to make sure that part-completed transactions are handled correctly in all circumstances.
-------------	--

To conclude the discussion of the APIs of `BlockingQueue` we should look at the third approach we mentioned above - methods that throw exceptions if the queue operation cannot immediately complete. These methods, `add()` and `remove()` are, frankly, problematic. This is for several reasons, not least of which is that the exceptions they throw on failure (`IllegalStateException` and `NoSuchElementException` respectively) are runtime exceptions and so do not need to be explicitly handled.

The problems with the exception-throwing API are deeper than just this, though. There is a general principle in Java that exceptions are to be used to deal with exceptional circumstances - i.e. those that a program does not normally consider to be part of normal operation.

The situation of an empty queue is, however, an entirely possible circumstance. So throwing an exception in response to it is a violation of the principle that is sometimes expressed as: "Don't use exceptions for flow control".

There is also the issue that exceptions are in general quite expensive to use, due to stack trace construction when the exception is instantiated and stack unwinding during the throw. It is good practice not to create an exception unless it is going to immediately be thrown.

For these reasons, we do recommend against using the exception-throwing form of the blocking queue APIs.

6.7.2 Using WorkUnit

The `Queue` interfaces are all generic - they're `Queue<E>`, `BlockingQueue<E>`, and so on. Although it may seem strange, it's sometimes wise to exploit this and introduce an artificial container class to wrap the items of work.

For example, if you have a class called `MyAwesomeClass` that represents the units of work that you want to process in a multithreaded application, then rather than having this:

```
BlockingQueue<MyAwesomeClass>
```

it can be better to have this:

```
BlockingQueue<WorkUnit<MyAwesomeClass>>
```

where `WorkUnit` (or `QueueObject`, or whatever you want to call the container class) is a packaging class that may look something like this:

```
public class WorkUnit<T> {
    private final T workUnit;

    public T getWork() {
        return workUnit;
    }

    public WorkUnit(T workUnit) {
        this.workUnit = workUnit;
    }

    // ... other methods elided
}
```

The reason for doing this is that this level of indirection provides a place to add additional metadata without compromising the conceptual integrity of the contained type (`MyAwesomeClass` in this example). In figure 5.8 we can see how the external metadata wrapper works.

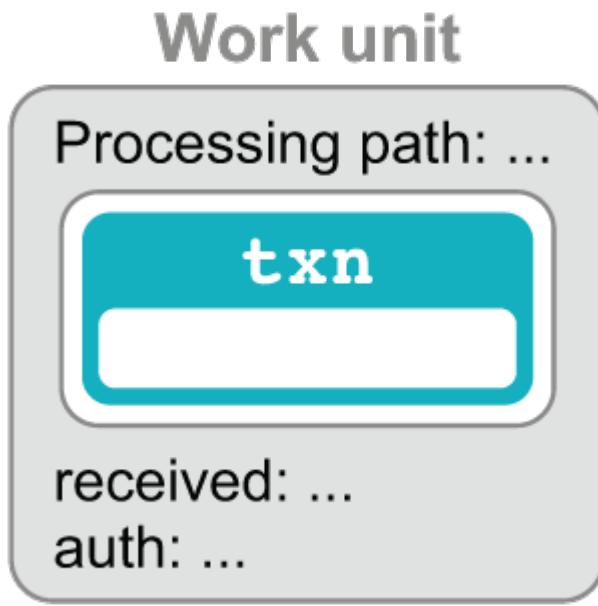


Figure 6.8 Use of a work unit as a metadata wrapper

This is surprisingly useful. Use cases where additional metadata is helpful are abundant. Here are a few examples:

- Testing (such as showing the change history for an object)
- Performance indicators (such as time of arrival or quality of service)
- Runtime system information (such as how this instance of `MyAwesomeClass` has been routed)

It can be much harder to add in this indirection after the fact. If you later discover that more metadata is needed in certain circumstances, it can be a major refactoring job to add in what would have been a simple change in the `WorkUnit` class.

Let's move on to discuss futures, which are a way of representing a placeholder for an in-progress (usually on another thread) task in Java.

6.8 Futures

The interface `Future` in `java.util.concurrent` is a very simple representation of an asynchronous task: it is a type that holds the result from a task that may not have finished yet but may do at some point in the future.

These are the primary methods on a `Future`:

- `get()` - This gets the result. If the result isn't yet available, will block until it is
- `isDone()` - This allows the caller to determine whether the computation has finished. It is non-blocking.
- `cancel()` - This allows the computation to be canceled before completion.

There's also a version of `get()` that takes a timeout, which won't block forever, in a similar manner to the blocking queue methods with timeouts that we met earlier.

The next snippet shows a sample use of a `Future` in a prime number finder:

Listing 6.5 Finding prime numbers using a Future

```
Future<Long> fut = getNthPrime(1_000_000_000);
Long result = null;

while (result == null) {
    try {
        result = fut.get(60, TimeUnit.SECONDS);
    } catch (TimeoutException tox) {
        // Timed out - better cancel the task
        System.err.println("Task timed out, cancelling");
        fut.cancel(true);
    } catch (InterruptedException e) {
        e.printStackTrace();
        break;
    } catch (ExecutionException e) {
        e.printStackTrace();
        break;
    }

    System.out.println("Still not found the billionth prime!");
}

System.out.println("Found it: " + result.longValue());
```

In this snippet, you should imagine that `getNthPrime()` returns a `Future` that is executing on some background thread (or even on multiple threads) - perhaps on one of the executor frameworks we'll discuss later in the chapter.

The thread running the snippet enters a `get-with-timeout` and blocks for up to 60 seconds for a response. If no response is received, then the thread loops and enters another blocking wait.

Even on modern hardware, this calculation may be running for a long time — so you may need to use the `cancel()` method after all (although the code as written does not provide any mechanism to cancel our request).

As a second example, let's consider non-blocking I/O. Figure 6.9 shows the future in action to allow us to use a background thread for I/O.

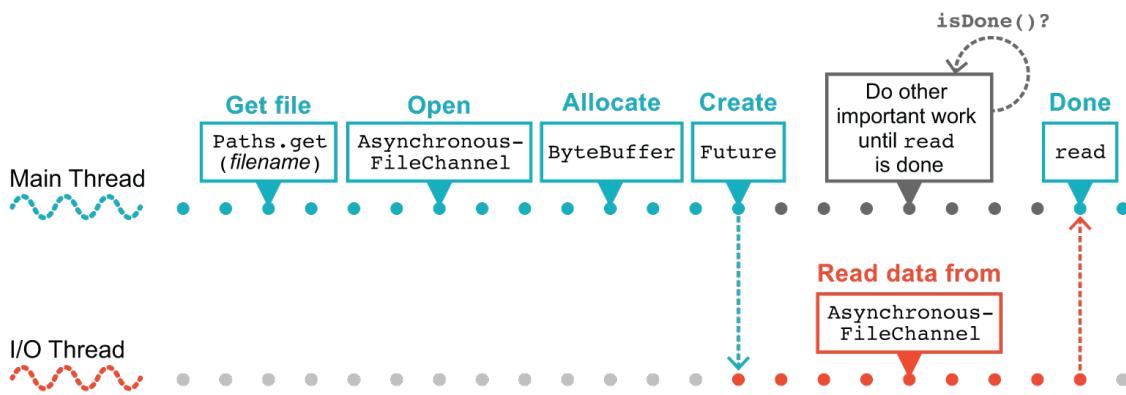


Figure 6.9 Using futures in Java

This API has been around for a while - it was introduced in Java 7 - and it allows the user to do non-blocking concurrency like this:

```
try {
    Path file = Paths.get("/Users/karianna/foobar.txt");

    var channel = AsynchronousFileChannel.open(file);           ①

    var buffer = ByteBuffer.allocate(1_000_000);
    Future<Integer> result = channel.read(buffer, 0);          ②

    BusinessProcess.doSomethingElse();                           ③

    var bytesRead = result.get();
    System.out.println("Bytes read [" + bytesRead + "]");
} catch (IOException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

- ① Open file asynchronously
- ② Request a read of up to 1 million bytes
- ③ Do something else
- ④ Get result when ready

This structure allows the main thread to `doSomethingElse()` whilst the I/O operation is proceeding on another thread - one which is managed by the Java runtime.

This is a useful approach, but requires support in the library that provides the capability. This can be somewhat limited - and what if we want to create our own asynchronous workflows?

6.8.1 `CompletableFuture`

The Java `Future` type is defined as an interface, rather than a concrete class. This means that any API that wants to use a future-based style has to supply a concrete implementation of `Future`.

These can be challenging for some developers to write and represents an obvious gap in the toolkit, so from Java 8 onwards a new approach to futures was included in the JDK. This is a concrete implementation of `Future` that enhances capabilities of futures and in some ways is more similar to futures in other languages (e.g. Kotlin and Scala).

The class is called `CompletableFuture` - it is a concrete type that implements the `Future` interface and provides additional functionality, and is intended as a simple building block for building asynchronous applications.

The central idea is that we can create instances of the `CompletableFuture<T>` type (it is generic in the type of the value that will be returned) and the object that is created represents the future in an *uncompleted* (or "unfulfilled") state.

Later, any thread that has a reference to the completable future can call `complete()` on it, and provide a value - this completes (or "fulfills") the future. The completed value is immediately visible to all threads that are blocked on a `get()` call. After completion, any further calls to `complete()` are ignored.

This means that the `CompletableFuture` cannot cause different threads to see different values. The future is either uncompleted or completed, and if it is completed then the value it holds is the value provided by the first thread to call `complete()`.

This is obviously not immutability - the state of the `CompletableFuture` does change over time. However, it only changes *once* - from uncompleted to completed. This means that there is no possibility of inconsistent state being seen by different threads.

NOTE

Java's `CompletableFuture` is similar to a promise as seen in other languages (such as Javascript), which is why we call out the alternative terminology of "fulfilling a promise" as well as "completing a future"

Let's look at an example, and implement `getNthPrime()` that we met earlier:

```

public static Future<Long> getNthPrime(int n) {
    var numF = new CompletableFuture<Long>(); ❶

    new Thread( () -> {
        long num = NumberService.findPrime(n);
        numF.complete(num); ❷
    } ).start(); ❸

    return numF;
}

```

- ❶ Create the completable future in an uncompleted state
- ❷ Create & start a new thread that will complete the future
- ❸ The actual calculation of the prime number

The method `getNthPrime()` creates an "empty" `CompletableFuture`, and returns this container object to its caller.

To drive this, we do need some code to call `getNthPrime()` - for example the code shown in Listing 5.5

One way to think about `CompletableFuture` is by analogy with client / server systems. The `Future` interface provides only a query method - `isDone()` and a blocking `get()` - this is playing the role of the client.

An instance of `CompletableFuture` plays the role of the server side - it provides full control over the execution and completion of the code that is fulfilling the future and providing the value.

In the example, `getNthPrime()` evaluates the call to the number service in a separate thread. When this call returns, we complete the future explicitly.

A slightly more concise way to achieve the same effect is to use the `CompletableFuture.supplyAsync()` method, passing a `Callable<T>` object representing the task to be executed. This call makes use of an application wide thread pool that is managed by the concurrency library.

```

public static Future<Long> getNthPrime(int n) {
    return CompletableFuture.supplyAsync(
        () -> NumberService.findPrime(n));
}

```

This concludes our initial tour of the concurrent data structures that are some of the main building blocks that provide the raw materials for developing solid multithreaded applications.

NOTE

We will have more to say about the `CompletableFuture` later in the book - specifically in the chapters that discuss advanced concurrency and the interplay with functional programming.

Next, we'll introduce the *executors* and threadpools that provide a higher-level and more convenient way to handle execution than the raw API based on `Thread`.

6.9 Tasks and execution

The class `java.lang.Thread` has existed since Java 1.0 - one of the original talking points of the Java language was built-in, language-level support for multithreading. It is powerful, and expresses concurrency in a form that is close to the underlying operating system support. However, it is a fundamentally low-level API for handling concurrency.

This low-level nature makes it hard for many programmers to work with correctly or efficiently. Other languages, that were released after Java, learned from Java's experience with threads and built upon them to provide alternative approaches. Some of those approaches have, in turn, influenced the design of `java.util.concurrent` and later innovations in Java concurrency.

In this case, our immediate goal is to have tasks (or work units) that can be executed without spinning up a new thread for each one. Ultimately, this means that the tasks have to be modeled as code that can be called rather than directly represented as a thread.

Then, these tasks can be scheduled on a shared resource - a pool of threads - that execute a task to completion, and then move on to the next task. Let's take a look at how we model these tasks.

6.9.1 Modeling tasks

In this section, we'll look at two different ways of modeling tasks — the `Callable` interface and the `FutureTask` class. We could also consider `Runnable` - but it is not always that useful, because the `run()` method does not return a value, and so therefore it can only perform work via side-effects.

One other aspect of the task modeling is important, but may not be obvious. It is the notion that if we assume that our thread capacity is finite, this means that the tasks must definitely complete in bounded time.

If there is the possibility of an infinite loop, then some tasks could "steal" an executor thread from the pool and this would reduce the overall capacity for all tasks from then on. Over time, this could eventually lead to exhaustion of the thread pool resource and no further work being possible.

As a result, we must be careful that any tasks we construct do actually obey the "terminate in finite time" principle.

CALLABLE INTERFACE

The `Callable` interface represents a very common abstraction. It represents a piece of code that can be called and returns a result. Despite being a straightforward idea, this is actually a subtle and powerful concept that can lead to some extremely useful patterns.

One typical use of a `Callable` is the lambda expression (or an anonymous implementation). The last line of this snippet sets `s` to be the value of `out.toString()`:

```
var out = getSampleObject();
Callable<String> cb = () -> out.toString();

String s = cb.call();
```

Think of a `Callable` as being a deferred invocation of the single method, `call()`, which the lambda provides.

FUTURETASK CLASS

The `FutureTask` class is one commonly used implementation of the `Future` interface, which also implements `Runnable`. As we'll see, this means that a `FutureTask` can be fed to executors. The API of `FutureTask` is basically that of `Future` and `Runnable` combined: `get()`, `cancel()`, `isDone()`, `isCancelled()`, and `run()`, although the last of these - the one that does the actual work - would be called by the executor, rather than directly by client code.

Two convenience constructors for `FutureTask` are provided: one that takes a `Callable` and one that takes a `Runnable` (which uses `Executors.callable()` to convert the `Runnable` to a `Callable`).

This suggests a flexible approach to tasks, allowing a job to be written as a `Callable` then wrapped into a `FutureTask` that can then be scheduled (and cancelled if necessary) on an executor, due to the `Runnable` nature of `FutureTask`.

The class provides a simple state model for tasks, and management of a task through that model. The possible state transitions are shown in Figure 6.10

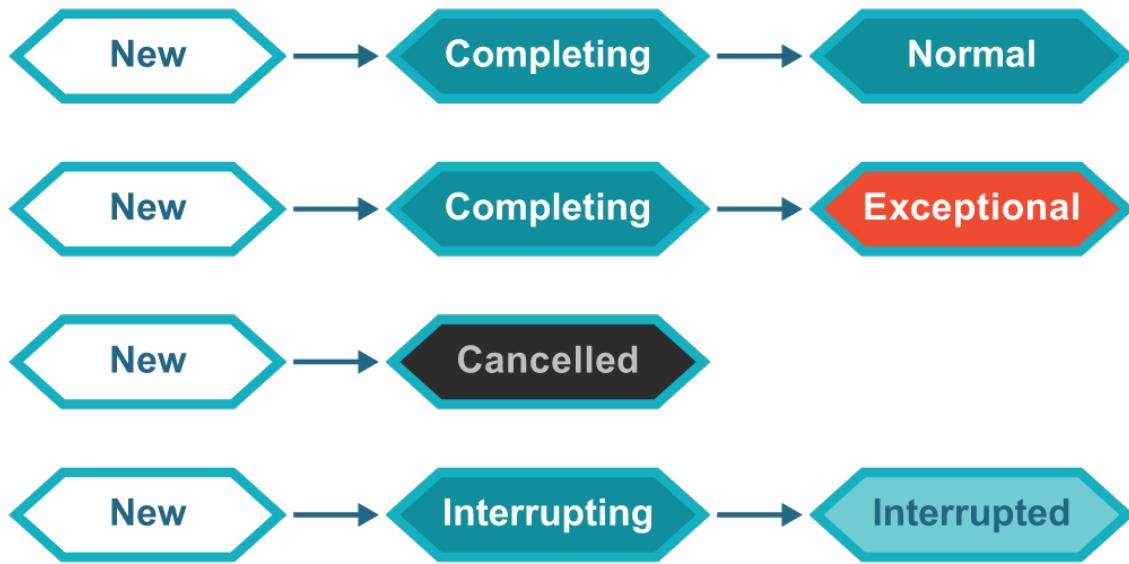


Figure 6.10 State model for tasks

This is sufficient for a wide range of ordinary execution possibilities.

Let's meet the standard executors that are provided by the JDK.

6.9.2 Executors

There are a couple of standard interfaces that are used to describe the threadpools present in the JDK. The first is `Executor` - which is very simple and defined like this:

```

public interface Executor {

    /**
     * Executes the given command at some time in the future. The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}

```

You should note that although this interface has only a single abstract method (i.e. it is a so-called SAM type), it is *not* tagged with the annotation `@FunctionalInterface`. It can still be used as the target type for a lambda expression, but it is not intended for use in functional programming.

In fact, the `Executor` is not widely used - far more common is the `ExecutorService` interface that extends `Executor` and adds `submit()` as well as several *lifecycle* methods, such as `shutdown()`.

To help the developer instantiate and work with some standard threadpools, the JDK provides the `Executors` class - which is a collection of static helper methods (mostly factories).

Four of the most commonly used methods are:

```
newSingleThreadExecutor()
newFixedThreadPool(int nThreads)
newCachedThreadPool()
newScheduledThreadPool(int corePoolSize)
```

To start with - let's look at each of these in turn. Later in the book, we will dive into some of the other, more complex, possibilities that are also provided.

6.9.3 Single threaded executor

The simplest of the executors is the single-threaded executor. This is essentially an encapsulated combination of a single thread and a task queue (which is a blocking queue).

Client code places an executable task onto the queue via `submit()`. The single execution thread then takes the tasks one at a time, and runs each to completion before taking the next task.

NOTE The executors are not implemented as distinct types, but instead represent different parameter choices when constructing an underlying threadpool.

Any tasks that are submitted whilst the execution thread is busy are queued until the thread is available. As this executor is backed by a single thread, then if the previously-mentioned "terminate in finite time" condition is violated, it means that no subsequently-submitted job will ever run.

NOTE This version of the executor is often useful for testing as it can be made more deterministic than other forms.

Here is a very simple example of how to use the single-threaded executor:

```
var pool = Executors.newSingleThreadExecutor();
Runnable hello = () -> System.out.println("Hello world");
pool.submit(hello);
```

The `submit()` call hands off the runnable task, by placing it on the executor's job queue. This means that job submission is non-blocking (unless the job queue is full).

However, care must still be taken - for example, if the main thread exits immediately, the submitted job may not have had time to be collected by the pool thread and so may not run. So

instead of exiting straightaway, it is wise to call the `shutdown()` method on the executor first.

The details can be found in the `ThreadPoolExecutor` class, but basically this method starts an *orderly shutdown* in which previously submitted tasks are executed, but no new tasks will be accepted. This effectively solves the issues we saw in Listing 5.4 about draining the queues of pending transactions.

NOTE

The combination of a task that loops infinitely and an orderly shutdown request will interact badly, resulting in a threadpool that never shuts down.

Of course, if the single-threaded executor was all that was needed, there wouldn't be a need to develop a deep understanding of concurrent programming and its challenges. So, we should also look at the alternatives that utilize multiple executor threads.

6.9.4 Fixed thread pool

The fixed thread pool, obtained via one of the variants of `Executors.newFixedThreadPool()`, is essentially the multiple-thread generalization of the single threaded executor. At creation time, the user supplies an explicit thread count, and the pool is created with that many threads.

These threads will be reused to run multiple tasks, one after another. The design prevents users having to pay the cost of thread creation. As with the single-threaded variant, if all threads are in use then new tasks are stored in a blocking queue until a thread becomes free.

This version of the threadpool is particularly useful if task flow is stable and known, and if all submitted jobs are roughly the same size, in terms of computation duration.

It is, once again, most easily created from the appropriate factory method:

```
var pool = Executors.newFixedThreadPool(2);
```

This will create an explicit thread pool backed by 2 executor threads. The two threads will take it turns to accept tasks from the queue in a non-deterministic manner. This means that even if there is a strict temporal (time-based) ordering of when tasks are submitted, there is no guarantee of which thread will handle a given task.

One consequence of this is that in a situation like that shown in Figure 6.11 then the tasks in the downstream queue cannot be relied upon to be accurately temporally ordered, even if the tasks in the upstream queue are.

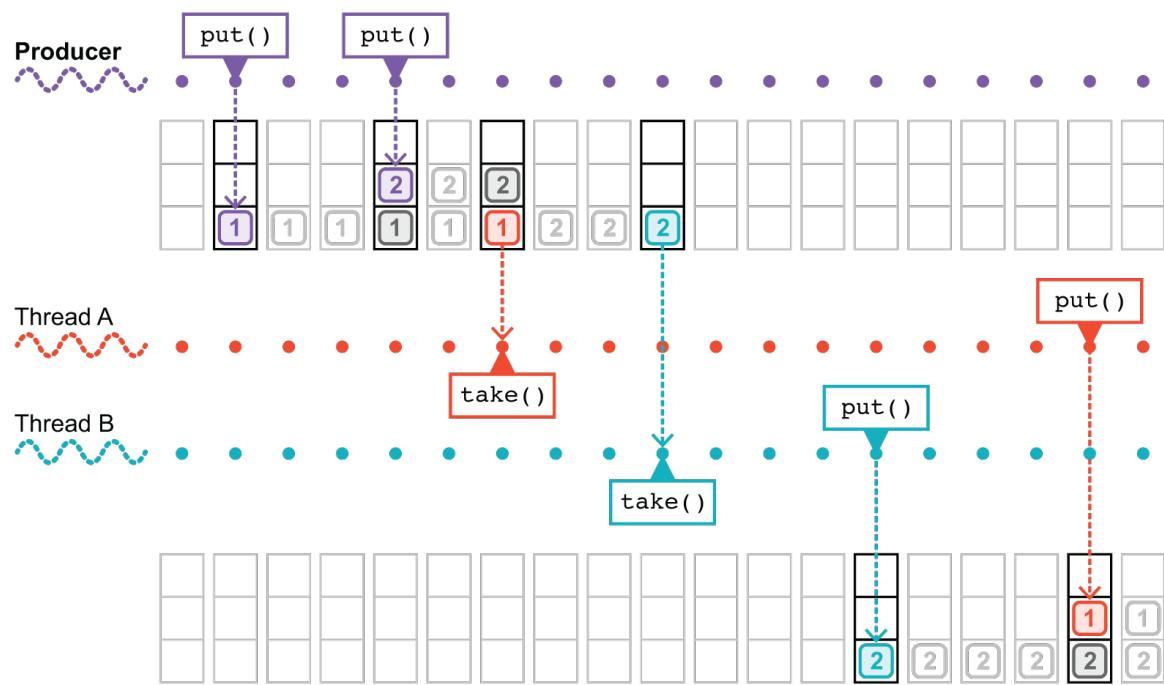


Figure 6.11 A threadpool and two queues

The fixed threadpool has its uses, but it is not the only game in town. For one thing, if the executor threads in it die, then they are not replaced. If the possibility exists of the submitted jobs throwing a runtime exception, then this can lead to the threadpool starving.

Let's look at another alternative, which makes different trade-offs but which can avoid this possibility.

6.9.5 Cached thread pool

The fixed threadpool is often used when the activity pattern of the workload is known, and fairly stable. However, if the incoming work is more uneven, or bursty, then a pool with a fixed number of threads is likely to be suboptimal.

The `CachedThreadPool` is an unbounded pool, that will reuse threads if they are available, but otherwise will create new threads as required to handle incoming tasks.

Threads are kept in the idle cache for 60 seconds and if they are still present at the end of that period they will be removed from the cache and destroyed.

```
var pool = Executors.newCachedThreadPool();
```

It is, of course, still very important that the tasks do actually terminate. If not, then the threadpool will, over time, create more and more threads and consume more and more of the machine's resources and eventually crash or become unresponsive.

The design of the `CachedThreadPool` should give better performance with small asynchronous tasks as compared to the performance achieved from fixed-size pools, but as always, if the effect is thought to be significant, proper performance testing much be undertaken.

6.9.6 ScheduledThreadPoolExecutor

The final example of an executor that we'll look at is a little bit different. This is the `ScheduledThreadPoolExecutor`, sometimes referred to as an STPE.

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(4);
```

Note that the return type, which we've explicitly called out here, is `ScheduledExecutorService`. This is different from the other factory methods, which return `ExecutorService`.

NOTE The `ScheduledThreadPoolExecutor` is a surprisingly capable choice of executor and can be used across a wide range of circumstances.

The scheduled service extends the usual executor service and adds a small amount of new capabilities: `schedule()` - which runs a 1-off task after a specified delay and two methods for scheduling periodic (i.e. repeating tasks) - `scheduleAtFixedRate()` and `scheduleWithFixedDelay()`.

The behaviour of these two methods is slightly different - `scheduleAtFixedRate()` will activate a new copy of the task on a fixed timetable (and it will do so whether or not previous copies have completed or not), whereas `scheduleWithFixedDelay()` will only activate a new copy of the task after the previous instance has completed, and the specified delay has elapsed.

Apart from the `ScheduledThreadPoolExecutor`, all the other pools we've met are obtained by choosing slightly different parameter choices for the quite general `ThreadPoolExecutor` class. For example, if we look at the definition of `Executors.newFixedThreadPool()`:

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

This is the purpose of the helper methods, of course, to provide a convenient way of accessing some standard choices for a threadpool without needing to engage with the full complexity of `ThreadPoolExecutor`.

Beyond the JDK, there are many other examples of executors and related threadpools, e.g. the `org.apache.catalina.Executor` class from the Tomcat web server.

6.10 Summary

In this chapter, we've addressed a subject that should be important to the well-grounded Java developer - the classes and concurrency primitives in `java.util.concurrent`.

Although we've looked at some theory, the most important part is the practical examples. Even if you just start with `ConcurrentHashMap` and the `Atomic` classes, you'll be using well-tested classes that can immediately provide real benefit to your code.

This should be your preferred toolkit for all new multithreaded Java code that you write. We've met these key building blocks and components for concurrent applications:

- Atomic integers
- Concurrent data structures, especially `ConcurrentHashMap`
- Blocking Queues and Latches
- Threadpools and executors

We've prepared the ground for you to begin using the classes of `java.util.concurrent` in your own code. This is the single most important takeaway from this chapter. We've seen how these components can be used to implement safe concurrent programming techniques, including:

- Inflexibility of `synchronized` locks
- Use of blocking queues for task handoff
- Using latches for consensus among a group of threads
- Partitioning execution into work units
- Job control, including safe shutdown

It's time to move on to the next big topic that will help you stand out as a Java developer. In the next chapter, you'll gain a firm grounding in the often-misunderstood subject of performance analysis.

Alternative JVM languages

This chapter covers

- Language zoology
- Why you should use alternative JVM languages
- Selection criteria for alternative languages
- How the JVM handles alternative languages

If you've used Java for any sizable amount of work, you've probably noticed that it can tend toward being a bit verbose and clumsy at times. You may even have found yourself wishing that things were different — easier somehow.

Fortunately, as you've seen in the last few chapters, the JVM is awesome! So awesome, in fact, that it provides a natural home for programming languages other than Java. In this chapter, we'll show you why and how you might want to start mixing another JVM programming language into your project.

In this chapter, we'll cover ways of describing the different language types (such as static versus dynamic typing), why you might want to use alternative languages, and what criteria to look for in choosing them. You'll also be introduced to the two languages (Kotlin and Clojure) that we'll cover in more depth throughout the remainder of this book.

8.1 Language zoology

Programming languages come in many different flavors and classifications. Another way of saying this is that there is a wide range of styles and approaches to programming that are embodied in different languages. Mastering these different styles is often easier when you understand how to classify the differences between languages.

NOTE These classifications are an aid to thinking about the diversity of languages. Some of these divisions are more clear-cut than others, but none of the classifying schemes is perfect. Different people have different ideas about how the classification should be laid out.

In recent years there has also been a trend for languages to add features from across the spectrum of possibilities. This means that it's often helpful to think of a given language as being "less functional" than another language, or "dynamically typed but with optional static typing when needed".

The classifications we'll cover are "interpreted versus compiled," "dynamic versus static," "imperative versus functional," and reimplementations of a language versus the original. In general, these classifications are tools for thinking about the space, rather than complete and precise academic schemes.

For example, we can say Java is a runtime-compiled, statically typed, imperative language with some functional features. It emphasizes safety, code clarity, backwards compatibility and performance, and it's happy to accept a certain amount of verbosity and *ceremony* (such as in deployment) to achieve those goals.

NOTE Different languages may have different priorities; for example, dynamically typed languages may emphasize deployment speed.

Let's get started with the interpreted versus compiled classification.

8.1.1 Interpreted vs. compiled languages

An interpreted language is one in which each step of the source code is executed as is, rather than the entire program being transformed to machine code before execution begins. This contrasts with a compiled language, which is one that uses a compiler to convert the human-readable source code into a binary form as an initial task.

This distinction has become less clear recently. In the early '90s, the divide was fairly clear: C/C++, FORTRAN and their friends were compiled languages, and Perl and Python were interpreted languages. But as we alluded to in chapter 1, Java has features of both compiled and interpreted languages. The use of bytecode further muddies the issue. Bytecode is certainly not human readable, but neither is it machine code.

For the JVM languages we'll study in this part of the book, the distinction we'll make is whether the language produces a class file from the source code, and executes that — or not. In the latter case, there will be an interpreter (probably written in Java) that's used to execute the source

code, line by line. Some languages provide both a compiler and an interpreter, and some provide an interpreter and a just-in-time (JIT) compiler that will emit JVM bytecode.

8.1.2 Dynamic vs. static typing

In languages with dynamic typing, a variable can contain different types at different times during a program's execution. As an example, let's look at a simple bit of code in a well-known dynamic language, JavaScript. This example should hopefully be comprehensible even if you don't know the language in detail:

```
var answer = 40;
answer = answer + 2;
answer = "What is the answer? " + answer;
```

The `var` keyword used here creates a new variable. In JavaScript's dynamic type system, this variable can contain a value of any type. This variable starts off set to `40`, which is, of course, a numeric value. We then add `2` to it, giving `42`. Then we change track slightly and make `answer` hold a string value. This is a very common technique in a dynamic language, and it causes no syntax errors.

The JavaScript interpreter is also able to distinguish between the two uses of the `+` operator. The first use of `+` is numeric addition — adding `2` to `40`, whereas in the following line the interpreter figures out from context that the developer meant string concatenation.

Let's try this trick again in Java using JShell:

```
jshell> var answer = 40;
answer ==> 40

jshell> answer = answer + 2;
answer ==> 42

jshell> answer = "What is the answer? " + answer;
| Error:
| incompatible types: java.lang.String cannot be converted to int
| answer = "What is the answer? " + answer;
| ^-----^
```

Boom. Even though the exact same source code looked legal in both languages, Java's *static type system* prevented the final line from working. This is because Java's `var` keyword does more than simply create the variable `answer`. As we learned about in section 1.3, Java's `var` also inferred the type of this new variable from the right-hand side of the expression. We didn't have to specify the type of `answer` explicitly, but Java's static type system assigns a type which never subsequently changes.

NOTE

The key point here is that dynamic typing tracks type information with the value a variable contains (for example, a number or a string), where static typing tracks the type with the variable definition instead.

Static typing can be a good fit for a compiled language because the type information is all about the variables, not the values in them. This allows reasoning about potential type system violations at compile time, before the code even has a chance to run.

Dynamically typed languages carry type information on the values held in variables. This provides a lot of flexibility, but means type violations (for example, "I thought this was a number but it's a string") happen during execution. This can lead to more runtime errors, which can be harder and more expensive to debug than compile-time errors.

8.1.3 Imperative vs. functional languages

Java is a classic example of an imperative language. Imperative languages can be thought of as languages that model the running state of a program as mutable data and issue a list of instructions that transform that running state. Program state is thus the concept that has center stage in imperative languages.

There are two main subtypes of imperative languages. Procedural languages, such as BASIC and FORTRAN, treat code and data as completely separate, and have a simple code-operates-on-data paradigm. The other subtype is object-oriented (OO) languages, where data and code (in the form of methods) are bundled together into objects. The program state in an object-oriented system is the state of all the objects in the program. In OO languages, additional structure is imposed to a greater or lesser degree by metadata (such as class information).

Functional languages take the view that computation itself is the most important concept. Functions operate on values, as in procedural languages, but instead of altering their inputs, functions are seen as acting like mathematical functions and return new values.

As illustrated in figure 8.1, functions are seen as "little processing machines" that take in values and output new values. They don't have any state of their own, and it doesn't really make sense to bundle them up with any external state. This means that the object-centered view of the world is somewhat at odds with the natural viewpoint of functional languages.

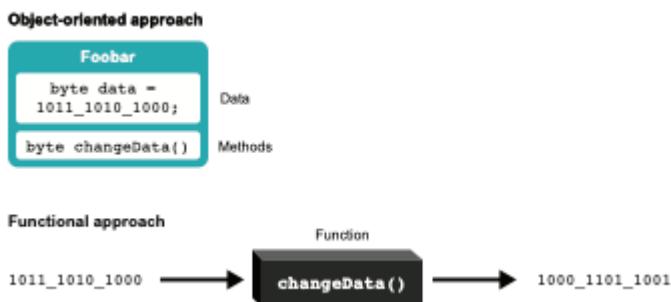


Figure 8.1 Imperative and functional languages

A key feature of functional languages is *first-class functions* - the ability to treat a function as a

value, assigning it to variables, passing it to other functions, even returning functions from other functions.

This is a great example of the feature spectrum we discussed earlier, as Java 8 added the lambda expression syntax which enables Java programmers to treat functions as value. However, as a recent addition the feature isn't used everywhere they could be in the platform, and older techniques for getting similar behavior such as the `Runnable` and `Callable` interfaces remain in use.

In the next two chapters we'll learn about different languages, and a key focus will be on how they support functional programming approaches. With Kotlin, we'll see how even an imperative language can be designed to smoothly support functional ideas. Then we'll look at Clojure, a much purer functional language which no longer centers object-orientation at all.

8.1.4 Reimplementation vs. original

Another important distinction between JVM languages is the division into those that are reimplementations of existing languages versus those that were specifically written to target the JVM. In general, languages that were specifically written to target the JVM provide a much tighter binding between their type systems and the native types of the JVM.

The following three languages are JVM reimplementations of existing languages:

JRuby is a JVM reimplementation of the Ruby programming language. Ruby is a dynamically typed OO language with some functional features. It's basically interpreted on the JVM, but recent versions have included a runtime JIT compiler to produce JVM bytecode under favorable conditions.

Jython was started in 1997 by Jim Hugunin as a way to use high-performance Java libraries from Python. It's a reimplementation of Python on the JVM, so it's a dynamic, mostly OO language. It operates by generating internal Python bytecode, then translating that to JVM bytecode. Sadly the project has seen little activity since 2015 and only supports Python 2.7, not the current Python 3.

Rhino was originally developed by Netscape, and later the Mozilla project. It provided an implementation of JavaScript on the JVM and shipped up through JDK 7. JDK 8 included a new JavaScript engine, *Nashorn* (The name "Nashorn" is a linguistic pun - it's the German word for "Rhino") but the increasing pace of JavaScript language changes forced its deprecation with JDK 11 and removal in JDK 15. Although no JavaScript implementation will ship directly with future JDKs, both may still be found independently ([Rhino from Mozilla](#), [Nashorn as an independent OpenJDK project](#) which intends to live on and should be supported on future JDKs.)

NOTE

The earliest JVM language? The earliest non-Java JVM language is hard to pin down. Certainly, Kawa, an implementation of Lisp, dates to 1997 or so. In the years since then, we've seen an explosion of languages, to the point that it's almost impossible to keep track of them.

A reasonable guess at time of writing is that there are at least 200 languages that target the JVM. Not all can be considered to be active or widely used (and some are really very niche), but the large number indicates that the JVM is a very active platform for language development and implementation.

NOTE

In the versions of the language and VM spec that debuted with Java 7, all direct references to the Java language have been removed from the VM spec. Java is now simply one language among many that run on the JVM—it no longer enjoys a privileged status.

The key piece of technology that enables so many different languages to target the JVM is the class file format, as we discussed in chapter 4. Any language that can produce a class file is considered a compiled language on the JVM.

Let's move on to discuss how polyglot programming came to be an area of interest for Java programmers. We'll explain the basic concepts, and why and how to choose an alternative JVM language for your project.

8.2 Polyglot programming on the JVM

The phrase *Polyglot programming on the JVM* was coined to describe projects using one or more non-Java JVM languages alongside a core of Java code. One common way to think about polyglot programming is as a form of separation of concerns. As you can see in figure 8.2, there are potentially three layers where non-Java technologies can play a useful role. This diagram is sometimes called the polyglot programming pyramid, and it's originally due to the work of [Ola Bini](#).

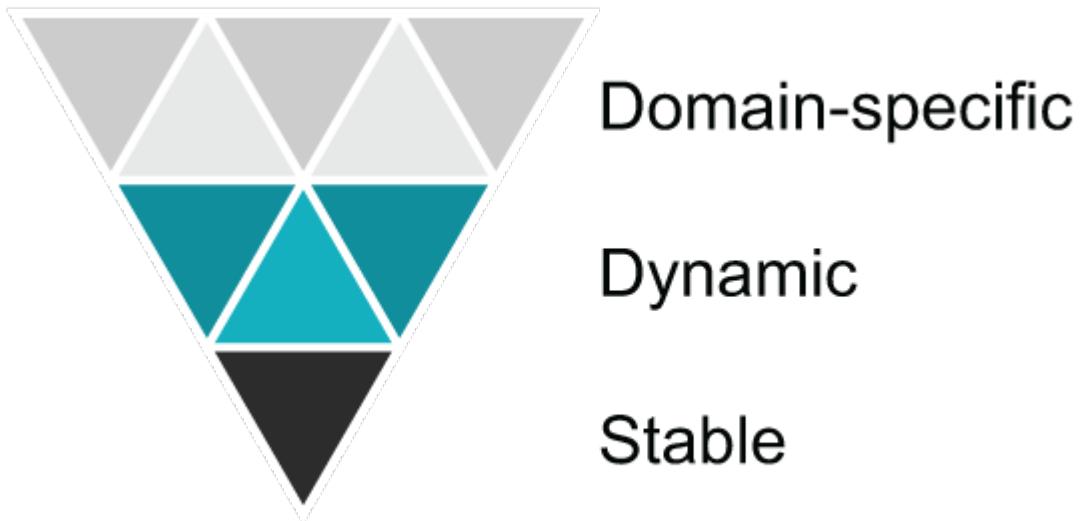


Figure 8.2 The polyglot programming pyramid

Within the pyramid, dependencies run in one direction - the stable layer is relatively independent, the dynamic layer uses the stable, and domain-specific code can pull in from both layers below it.

Defining these layers in a given system isn't always easy - there are gray areas, and not all systems fit perfectly. However, it's a useful tool to identify the seams where different parts of the system have different needs and could benefit from different languages.

The stable layer contains core APIs and abstractions for your system. Type safety, thorough testing, and performance are all critical.

The dynamic layer uses the stable layer's abstractions to create a working system. This may include code such as how a system exposes itself over HTTP or interacts with other backend systems. Issues like compilation time and flexibility may make it worth considering a different language at the dynamic layer.

The domain-specific layer handles application-specific concerns such as presentation, customization of rules and processing, or CI/CD. This code is all about specific aspects of the application domain, and may benefit from language choices that would be constraining in other layers.

NOTE The secret of polyglot programming. Polyglot programming makes sense because different pieces of code have different lifetimes. A risk calculation engine in a bank may last for five or more years. JSP pages for a website could last for a few months. The most short-lived code for a startup could be live for just a few days. The longer the code lives, the closer to the stable layer of the pyramid it is.

Table 8.1 Three layers of the polyglot programming pyramid

Name	Description	Examples
Domain-specific	Domain-specific language. Tightly coupled to a specific part of the application domain.	Apache Camel, DSLs, Drools, Web templating
Dynamic	Rapid, productive, flexible development of functionality.	Clojure, Groovy, JRuby
Stable	Core functionality, stable, well-tested, performant.	Java, Kotlin, Scala

As you can see, there are patterns in the layers — the statically typed languages tend to gravitate toward tasks in the stable layer. Conversely, the technologies intended for more of a specific purpose tend to be well-suited to roles at the top of the pyramid.

Let's dig a little deeper to look at why Java isn't the best choice for everything in the pyramid. We'll begin by discussing why you should consider a non-Java language, then we'll cover some of the major criteria to look at in choosing a non-Java language for your project.

8.2.1 Why use a non-Java language?

Java's nature as a general-purpose, statically typed, compiled language provides many advantages. These qualities make it a great choice for implementing functionality in the stable layer. But these same attributes become a burden in the upper tiers of the pyramid. For example,

- Recompilation is laborious
- Static typing can be inflexible
- Deployment is a heavyweight process
- Java's syntax can be rigid and isn't a natural fit for producing DSLs

The recompilation and rebuild time of a Java project often reaches the 90 seconds to 2 minutes mark. This is a long enough to seriously break a developers' flow, and it's a bad fit for developing code that may live in production for only a few weeks.

JAVA'S RIGID SYNTAX

The Java language has a very rigid grammar. The fundamental language components are the keywords that are supplied. You cannot "make up new syntax" or create any new form that could be mistaken for a keyword.

The programmer can create new classes - and the capabilities of those classes consist of storing state in fields and calling methods on classes or objects. However, this is as far as it goes - the programmer cannot create anything that resembles a control structure.

In other words, a field access will always look like:

```
anObject.someField
AClass.someStaticField
```

and a method call will always look like:

```
anObject.someMethod(params)
AClass.someStaticMethod(params)
```

In Java, the method parameters are never optional (unlike in some other languages, such as Kotlin) - so even the distinction between field access and methods calls cannot be blurred.

This means that we cannot create a `when` construct that looks like a keyword. The best we can do is something like this:

```
import static when.When.when;

...
when(value, () -> {
    // action to be taken
});
```

which is, of course, a long way from:

```
when(value) {
    // action to be taken
}
```

This lack of redefinable syntax also shows up when trying to use Java to make DSLs. We'll see how our non-Java languages handle this issue in the next 2 chapters.

Overall, one pragmatic solution is to play to Java's strengths. Take advantage of its rich API and library support to do the heavy lifting for the application down in the stable layer.

Even within the stable layer you may find there are reasons a language other than Java may be desirable. For example,

- Java's verbosity can be off-putting for some developers, and it can hide certain classes of bugs
- While Java increasingly supports functional programming, there remain limits in the ease of applying some patterns
- Other languages present alternatives for concurrency that are difficult to implement directly in Java (coroutines in Kotlin, actors in Clojure)

NOTE

Even if you choose another language to use in your stable layer for features it supports, you shouldn't throw out working code just to rewrite so the languages match. Consider using the new language for new features, or for low-risk areas we'll talk about identifying later in this chapter.

At this point, you may be asking yourself, "What type of programming challenges fit inside these layers? Which languages should I choose?" A well-grounded Java developer knows that there is no silver bullet, but we do have criteria to consider when evaluating your choices.

8.2.2 Up-and-coming languages

For the rest of the book, we've picked two languages that we see having great potential longevity and influence. These are two of the languages on the JVM (Kotlin and Clojure) that already have well-established mind share among polyglot programmers. So why are these languages gaining traction? Let's look at each in turn.

KOTLIN

Kotlin is an imperative, statically typed, OO language from JetBrains (makers of the IntelliJ IDE). It aims to address the most common complaints about Java, while keeping a familiar development environment. Kotlin is a compiled language and has a high degree of compatibility beyond the basics that just running on the JVM provides.

Key features in Kotlin include concise syntax, null safety and built-in support for coroutines, an alternate concurrency mechanism from Java's traditional threading model. A number of features from Kotlin have found their way back into Java in recent releases, confirming the value those changes presented to developers.

While it has established itself as a key JVM language option in multiple areas, Kotlin has shown particular success in mobile, with the Android platform adopting it as the recommended language in 2019. But the wide range of convenience and safety improvements for developers are worth considering regardless where your JVM is running. Chapter 9 introduces gives an introduction to Kotlin.

CLOJURE

Clojure, designed by Rich Hickey, is a language from the Lisp family. It inherits many syntactic features (and lots of parentheses) from that heritage. It's a dynamically typed, functional language, as is usual for Lisps. It's a compiled language, but usually distributes code in source form — for reasons we'll see later. It also adds a significant number of new features (especially in the arena of concurrency) to its Lisp core.

Lisps are usually seen as experts-only languages. Clojure is somewhat easier to learn than other Lisps, yet still provides the developer with formidable power (and also lends itself very nicely to the test-driven development style). But it's likely to remain outside the mainstream, being primarily used by enthusiasts and for specialized jobs (for example, some financial applications find its combination of features very appealing).

Clojure is best thought of as sitting in the dynamic layer, but due to its concurrency support and other features can be seen as capable of performing many of the roles of a stable layer language. Chapter 9 provides an introduction to Clojure.

8.2.3 Languages we could have picked but didn't

As noted before, a huge variety of languages exist that we could look at. Here's a little more about a few other contenders that may be practical for you to look at more deeply yourself.

GROOVY

The Groovy language was invented by James Strachan in 2003. It's a dynamic, compiled language with syntax very similar to Java's, but more flexible. It's widely used for scripting - it was the original language used by the Gradle build tool. It's often the first non-Java language that developers or teams investigate on the JVM. Groovy can be seen as sitting in the dynamic layer and is also known for being great for building DSLs.

We've chosen not to cover Groovy in more detail as its seen declining mindshare in the prototyping and application use cases in the face of improving frameworks and other languages.

SCALA

Scala is an OO language that also supports many aspects of functional programming. It traces its origins to 2003, when Martin Odersky began work on it, following his earlier projects related to generics in Java. It's a statically typed, compiled language like Java, and it performs a large amount of type inference. This means that it often has the feel of a dynamic language.

Scala learned a great deal from Java, and its language design "fixes" several common annoyances with Java. However, Scala has ended up with a very large set of features, though, and a much more advanced type system as compared to Java's.

This means that it can be complicated to program and is not easy to learn thoroughly. As such we've chosen to focus on Kotlin for developers who just want improvements to the state of the Java language.

GraalVM

Oracle Labs have produced [GraalVM](#), which they describe as a polyglot virtual machine and platform, which is partly derived from Java and JVM codebases.

The current release includes the capability to run Java and other JVM languages (as bytecode) as well as support for JavaScript and LLVM bitcode, with beta support for Ruby, Python, R and WASM.

The overall platform comprises a number of components:

- Java HotSpot VM
- A Node.js JavaScript runtime environment
- LLVM runtime to execute LLVM bitcode
- Graal - a JIT compiler written in Java
- Truffle - a toolkit and API for building language interpreters
- SubstrateVM - a lightweight execution container for native images

Within a GraalVM project, languages can be very freely bridged to each other and the aim is to allow components implemented in different technologies to be combined and used in a single application process.

This is a very different approach to polyglot programming, but it is close enough to the subject of interest that we wanted to at least mention it.

NON-JVM LANGUAGES

This chapter focuses on alternative languages that run on the JVM. It's worth admitting, though, that sometimes the polyglot programmer may have a reason part of their system needs to leave the JVM behind entirely.

Examples of technologies which have broader support outside the JVM:

- native system code (C or Rust)
- machine learning (Python)
- run in a user's web browser (JavaScript)

While there are JVM based approaches for many of these, it's worth taking stock of the maturity of alternatives and the composition of our team before trying to keep every line of code entirely on the JVM.

Now that we've outlined some possible choices, let's discuss the issues that should drive your decision of which language to choose.

8.3 How to choose a non-Java language for your project

Once you've decided to experiment with non-Java languages in your project, you need to identify which parts of your project naturally fit into the stable, dynamic, or domain-specific layers. Table 8.2 highlights tasks that might be suitable for each layer.

Table 8.2 Project areas suited for domain-specific, dynamic, and stable layers

Name	Example problem domains
Domain-specific Domain-specific areas often benefit from readability by experts who may not know Java. Software lifecycle tooling also frequently has domain-specific languages and configuration.	Build, continuous integration, continuous deployment Dev-ops Business rules modeling
Dynamic Dynamic layers of the system may benefit from greater flexibility and speed of development available in other languages. This may be especially true for tooling which is internally facing (testing and administrative).	Rapid web development Prototyping Interactive administrative and user consoles Scripting Testing
Stable Stable layer code expresses a system's core abstractions. More rigorous type safety and testing are worth the additional developer overhead.	Concurrent code Application containers Core business functionality

As you can see, there is a wide range of use cases for alternative languages. But identifying a task that could be resolved with an alternative language is just the beginning. You next need to evaluate whether using an alternative language is appropriate. Here are some useful criteria that we take into account when considering technology stacks:

- Is the project area low-risk?
- How easily does the language interoperate with Java?
- What tooling support (for example, IDE support) is there for the language?
- How steep is the learning curve for this language?
- How easy is it to hire developers with experience in this language?

Let's dive into each of these areas so you get an idea of the sorts of questions you need to be asking.

8.3.1 Is the project area low-risk?

Let's say you have a core payment-processing rules engine that handles millions of transactions a day. This is a stable piece of Java software that has been around for over seven years, but there aren't a lot of tests, and there are plenty of dark corners in the code. The core of the payment-processing engine is clearly a high-risk area to bring a new language into, especially when it's running successfully and there's a lack of test coverage and of developers who fully understand it.

But there's more to a system than its core processing. For example, this is a situation where better tests would clearly help. Kotlin has a number of good options, including [Spek framework](#) and [Kotest](#) that leverage the language to enable clear, readable specifications without the typical JUnit boilerplate. Or perhaps your rules engine would benefit from *property testing*, where tests are written to validate conditions against generated inputs and Clojure's [test.check](#) would be a valuable tool in the mix.

Or suppose you need to build a web console so that the operations users can administer some of the noncritical static data behind the payment-processing system. The development team members already know Struts and JSF, but don't feel any enthusiasm for either technology. This is another low-risk area to try out a new language and technology stack. Spring Boot with Kotlin would be one obvious choice.

By focusing on a limited pilot in an area that is low-risk, there's always the option of terminating the project and porting to a different delivery technology without too much disruption if the new technology stack isn't a good fit.

8.3.2 Does the language interoperate well with Java?

You don't want to lose the value of all of that great Java code you've already written! This is one of the main reasons organizations are hesitant to introduce a new programming language into their technology stack. But with alternative languages that run on the JVM, you can turn this on its head, so it becomes about maximizing your existing value in the codebase and not throwing away working code.

Alternative languages on the JVM are able to cleanly interoperate with Java and can, of course, be deployed on a preexisting environment. This is especially important to avoid impacting to whoever owns deployment, whether a production management team or DevOps folks in your own team. By using a non-Java JVM language as part of your system, you retain your organization's operational expertise. This can help alleviate worries and reduces risk around supporting the new solution.

NOTE **DSLs are typically built using a dynamic (or, in some cases, static) layer language, so many of them run on the JVM via the languages that they were built in.**

Some languages interoperate with Java more easily than others. We've found that most popular JVM alternatives (such as Kotlin, Clojure, JRuby, Groovy, and Scala) all have good interoperability with Java (and for some of the languages, the integration is excellent, almost completely seamless). If you're a really cautious shop, it's quick and easy to run a few experiments first, and make certain that you understand how the integration can work for you.

Let's take Kotlin, for example. You can import Java packages directly into its code via the familiar import statement. From here you could easily write a small Kotlin script, or even use the interactive Kotlin shell, to poke at your Java model objects and see what the interop surfaces will look like.

We'll talk specifically about Java interoperability in the language chapters that are coming up next.

8.3.3 Is there good tooling and test support for the language?

Most developers underestimate the amount of time they save once they've become comfortable in their environment. Their powerful IDEs and build and test tools help them to rapidly produce high quality software. Java developers have benefited from great tooling support for years, so it's important to remember that other languages may not be at quite the same level of maturity.

Some languages (such as Kotlin) have had longstanding IDE support for compiling, testing, and deploying the end result. Other languages may have tooling that hasn't matured as fully.

A related issue is that when an alternative language has developed a powerful tool for its own use (such as Clojure's awesome Leiningen build tool), the tool may not be well adapted to handle other languages. This means that the team will need to think carefully about how to divide up a project, especially for deployment of separate but related components.

8.3.4 How hard is the language to learn?

It always takes time to learn a new language, and that time only increases if the paradigm of the language isn't one that your development team is familiar with. Most Java development teams will be comfortable picking up a new language if it's object oriented with a C-like syntax (for example, Kotlin).

It gets harder for Java developers as they move further away from this paradigm. At the extreme of the popular alternative languages, a language such as Clojure can bring incredibly powerful

benefits, but can also represent a significant retraining requirement for development teams as they learn Clojure’s functional nature and Lisp syntax.

One alternative is to look at the JVM languages that are reimplementations of existing languages. Ruby and Python are well-established languages, with plenty of material available for developers to use to educate themselves. The JVM incarnations of these languages could provide a sweet spot for your teams to begin working with an easy-to-learn non-Java language.

8.3.5 Are there lots of developers using this language?

Organizations have to be pragmatic; they can’t always hire the top 2 percent (despite what their advertising might say), and their development teams will change throughout the course of a year. Some languages, such as Kotlin or Scala, are becoming well-established enough that there is a pool of developers to hire from. But a language such as Clojure may present more difficulties.

NOTE

A warning about the reimplemented languages: many existing packages and applications written in Ruby, for example, are only tested against the original C-based implementation. This means that there may be problems when trying to use them on top of the JVM. When making platform decisions, you should factor in extra testing time if you’re planning to leverage an entire stack written in a reimplemented language.

Again, the reimplemented languages (JRuby, Jython, and so on) can potentially help here. Few developers may have JRuby on their CV, but as it’s just Ruby on the JVM, there’s actually a large pool of developers to hire from — a Ruby developer familiar with the C version can learn the differences induced by running on the JVM very easily.

We now have a set of questions to ask when choosing an alternative language, and an overview of some available options. At this point, it’s worth a deeper understanding of how the JVM supports multiple languages. This reveals the roots of some design choices and limitations in alternative languages on the JVM.

8.4 How the JVM supports alternative languages

There are two possible ways that a language can run on the JVM:

- Have a source code compiler that emits class files
- Have an interpreter that is implemented in JVM bytecode

In both cases, it’s usual to have a runtime environment that provides language-specific support for executing programs. Figure 8.3 shows the runtime environment stack for Java and for a typical non-Java language.

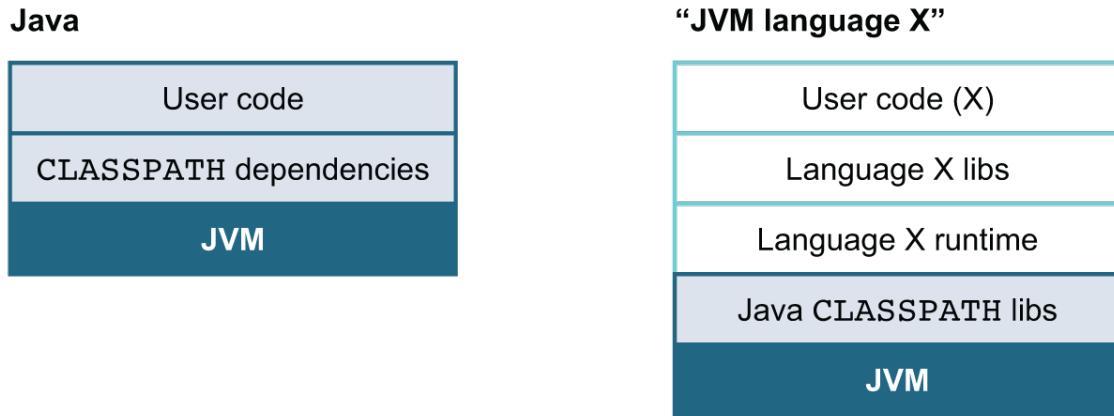


Figure 8.3 Non-Java language runtime support

These runtime support systems vary in complexity, depending on the amount of hand holding that a given non-Java language requires at runtime. In almost all cases, the runtime will be implemented as a set of JARs or modules that an executing program needs to have on its classpath. In the interpreted case, the interpreter will bootstrap as program execution starts, and then read in the source file to be executed.

8.4.1 Performance

One question developers often ask about different languages is "how do they perform relative to each other?". Whilst superficially attractive, this question is not simple to answer and in fact is not actually all that meaningful.

As we saw in Chapter 7, the well-grounded developer knows that performance is driven by measurement. Measurement is done on individual programs, not on the abstract notion of a programming language. Treat any claim that language X "performs better" than Y without accompanying, reliable data as suspect.

However, in practice, some overall performance characteristics of a JVM language can broadly be determined by how the language is implemented. A compiled language is just bytecode at runtime and will be JIT-compiled in just the same way as Java is. An interpreted language will have very different performance behavior - as the code that gets JIT-compiled is the interpreter, and not the program itself.

NOTE Some languages (e.g. JRuby) have a hybrid strategy - they have an interpreter for scripts but can also dynamically compile individual source methods to JVM bytecode - which can then be compiled to machine code by the JVM's JIT compilers.

In this book, our focus is on compiled languages. The interpreted languages — such as Rhino —

are mentioned for completeness, but we won't spend too much time on them. We therefore expect that performance will be broadly similar between the languages that we're considering. For a more detailed answer, a detailed analysis of a specific program or workload should be undertaken.

In the rest of this section, we'll discuss the need for runtime support for alternative languages (even for compiled languages) and then talk about compiler fictions — language-specific features that are synthesized by the compiler and that may not appear in the low-level bytecode.

8.4.2 Runtime environments for non-Java languages

One simple way to measure the complexity of the runtime environment that a particular language requires is to look at the size of the JAR files that provide the implementation of the runtime. Using this as a metric, we can see that Clojure is a relatively lightweight runtime, whereas JRuby is a language that requires more support.

This isn't a completely fair test, as some languages bundle much larger standard libraries and additional functionality into their standard distributions than others. However, it can be a useful (if rough) rule of thumb.

In general, the purpose of the runtime environment is to help the type system and other aspects of the non-Java language achieve the desired semantics. Alternative languages don't always have exactly the same view as Java about basic programming concepts.

For example, Java's approach to OO isn't universally shared by other languages. In Java, all objects that are instances of a particular class all have exactly the same set of methods on them - and that set is fixed at compile time.

In Ruby, on the other hand, an individual object instance can have additional methods attached to it at runtime that were not known when the class was defined and that aren't necessarily defined on other instances of the same class.

NOTE

The `invokedynamic` bytecode was in fact originally added to the JVM in order to facilitate the efficient implementation of these types of language features.

This ability to dynamically add methods (which is somewhat confusingly called "open classes") needs to be replicated by the JRuby implementation. This is only possible with some advanced support from the JRuby runtime.

8.4.3 Compiler fictions

Certain language features are synthesized by the programming environment and high-level language, and aren't present in the underlying JVM implementation. These are referred to as *compiler fictions*.

NOTE

It helps to have some knowledge of how these features are implemented — otherwise you can find your code running slowly, or in some cases even crashing the process. Sometimes the environment has to do a lot of work to synthesize a particular feature.

Other examples in Java include checked exceptions and inner classes (which are always converted to top-level classes with specially synthesized access methods if necessary, as shown in figure 8.4). If you've ever looked inside a JAR file (using `jar tvf`) and seen a load of classes with \$ in their names, then these are the inner classes unpacked and converted to "regular" classes.

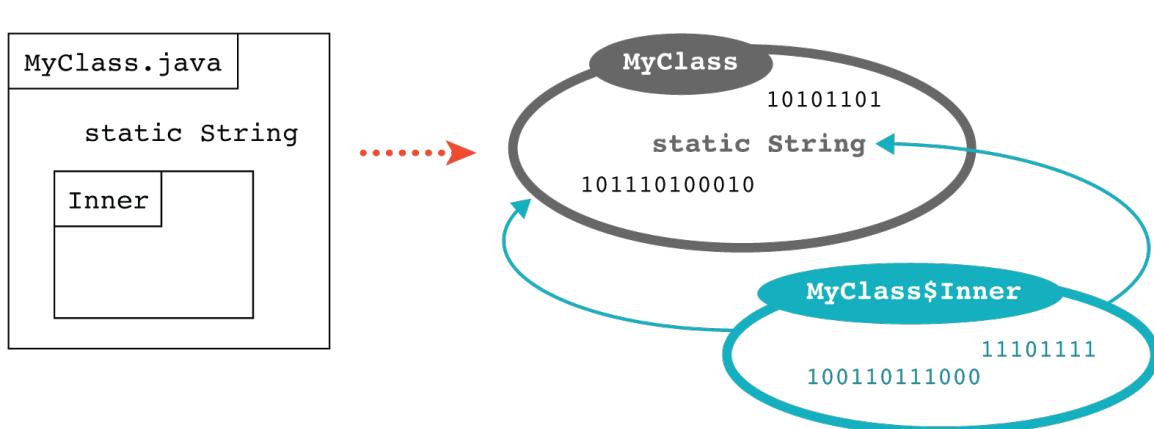


Figure 8.4 Inner classes as a compiler fiction

Alternative languages also have compiler fictions. In some cases, these compiler fictions even form a core part of the language's functionality.

In section 8.2, we introduced the key concept of *first-class functions* in functional programming — that functions should be values that can be put into variables. All the non-Java languages in part 3 of this book supported this feature long before Java added lambda expressions. How did they accomplish this when the JVM only handles classes as the smallest unit of code and functionality?

The original solution to this discrepancy between source code and JVM bytecode is to remember that objects are just bundles of data along with methods to act on that data. Imagine an object

with no state and just one method — for example, a simple anonymous implementation of Java’s `Callable`. It wouldn’t be at all unusual to put such an object in a variable, pass it around, and then invoke its `call()` method later, like this:

```
Callable<String> myFn = new Callable<String>() {
    @Override
    public String call() {
        return "The result";
    }
};

System.out.println(myFn.call());
```

NOTE

The `myFn` variable in this example is an anonymous type, so it will show up after compilation as something like `NameOfEnclosingClass$1.class`. The class numbers start at 1 and go up for each anonymous type the compiler encounters. If they’re dynamically created, and there are a lot of them (as sometimes happens in languages like JRuby), this can place pressure in the off-heap memory where the definitions of classes are stored.

Java lambda expressions don’t actually use this anonymous type approach, but instead are built on a general JVM feature called `invokedynamic` which we’ll discuss in detail in Chapter 16. Alternate languages are moving from their specialized implementations to use `invokedynamic` too. It’s an interesting case of compiler fictions influencing the development of platform realities.

For another example - in the next chapter we will meet Kotlin’s *data classes* - a language feature that helps to lower the amount of typing and *ceremony* required when declaring a class that is “just a dumb bunch of fields”. In Kotlin as it exists today this is a compiler fiction - but Java is evolving to add a feature called *records* that may eventually provide an alternative basis for Kotlin to build data classes upon.

8.5 Summary

Alternative languages on the JVM have come a long way. They can now offer better solutions than Java for certain problems while retaining compatibility with existing systems and investments made in Java technology. This means that even for Java shops, Java isn’t always the automatic choice for every programming task.

Understanding the different ways languages can be classified (static versus dynamic, imperative versus functional, and compiled versus interpreted) gives you the foundation for being able to pick the right language for the right task.

For the polyglot programmer, languages fall roughly into three programming layers: stable, dynamic, and domain-specific. Languages such as Java and Kotlin are best used for the stable layer of software development, whereas others, such as Clojure, are more suited to tasks in the

dynamic or domain-specific realms.

Certain programming challenges fit well into particular layers, such as rapid web development in the dynamic layer or modeling enterprise messaging in the domain-specific layer.

It's worth emphasizing again that the core business functionality of an existing production application is almost never the correct place to introduce a new language. The core is where high-grade support, excellent test coverage, and a proven track record of stability are of paramount importance. Rather than start here, choose a low-risk area for your first deployment of an alternative language.

Always remember that each team and project has its own unique characteristics that will impact the decision of which language to choose. There are no universal right answers here. When choosing to implement a new language, managers and senior techs must take into account the nature of their projects and team.

There are two alternative languages on the JVM that we see leading the pack: Kotlin and Clojure. By the time you finish this book, you'll have learned the basics of these promising alternative languages on the JVM and will have expanded your programming toolkit in interesting new directions.

In the next chapter, you'll learn about the first of these — Kotlin.

A large, light gray number '9' is positioned above the word 'Kotlin'. The '9' has a thick, rounded stroke. To its right, the word 'Kotlin' is written in a bold, dark blue sans-serif font.

This chapter covers

- Why Kotlin?
- Convenience and conciseness
- Safety
- Concurrency
- Java Interoperability

9.1 Why Kotlin?

Kotlin is a language created by [JetBrains](#), makers of the popular IntelliJ IDE. Announced publicly in 2011, Kotlin aims to fill language gaps they felt developing in Java, without the friction they saw with other existing JVM languages.

Kotlin was open-sourced the following year, and reached 1.0 - with guaranteed levels of support and maintenance from JetBrains - in 2016. Since then, it has gone on to become the recommended language for the Android platform and gathered a solid following in other JVM coding circles. It has also reached beyond the JVM, supporting JavaScript and native backends as well.

As alternate languages go, Kotlin provides many quality of life improvements over Java, while not radically changing the entire world. Its focus on convenience, safety, and solid interop has given it a great story for incremental usage in existing Java projects. With first-class support in their IntelliJ IDE, turning a file from Java to Kotlin is often just a click away.

It's worth noting that some features originally only available in Kotlin have made their way back into Java. A great example of this is Kotlin script - Kotlin can run a source file directly, typically

with the `kts` extension, without the developer asking to compile. If this sounds like Java 11's single-file feature that we showed off in Chapter 1, you're not wrong!

Let's start seeing what Kotlin can provide for us.

9.1.1 Installing

If you use IntelliJ, Kotlin is provided via a plugin. With that installed you can start writing code in Kotlin straight away just like any other supported language in the IDE.

For those who are more inclined to bare-bones setups Kotlin [also provides](#) a command-line compiler (`kotlinc`) and interactive shell (`kotlin`).

Adding Kotlin to an existing project requires updating your build scripts. Chapter 11 will get you more familiar with these systems, but for now you can refer to Kotlin's excellent documentation to get you started with either [Maven](#) or [Gradle](#).

With Kotlin ready to run a good place to start exploring is in how its basic features work and improve on Java.

9.2 Convenience and conciseness

Java has a reputation for being verbose. While it retains a lot of visual similarity with Java, Kotlin relentlessly streamlines the code you have to write.

9.2.1 Starting with less

One example of this is that simple character, the semicolon. Kotlin doesn't require semicolons for line ending, allowing the typical newlines to stand in their place. While semicolons are allowed - and in fact required if you want to put multiple statements on a single line for instance - most often they aren't needed.

Kotlin took advantage of its blank slate to alter other defaults that would be difficult to change in Java. While Java only imports `java.lang` by default, Kotlin provides these packages everywhere:

- `java.lang.*`
- `kotlin.*`
- `kotlin.annotation.*`
- `kotlin.collections.*`
- `kotlin.comparisons.*`
- `kotlin.io.*`
- `kotlin.ranges.*`
- `kotlin.sequences.*`
- `kotlin.text.*`
- `kotlin.jvm.*`

It's the rare program that doesn't need collections, text, or IO, so these inclusions save a lot of needless importing.

A spot where Java's verbosity often shows up is with types - and we saw in Chapter 1 how the `var` keyword cuts down on repetition of type information. Kotlin has had this style of type inference since the beginning - although it is far from the only source for the idea.

9.2.2 Variables

When introducing a variable, Kotlin even uses the same keyword as newer Java: `var`. It will infer from the expression on the right hand side what type to use for the variable.

```
var i = 1          ①
var s = "String"  ②
```

- ① `i` is type `kotlin.Int`
- ② `s` is type `kotlin.String`

Unlike Java, though, in Kotlin `var` is not just a shortcut. If you want to explicitly call out the type of the variable, `var` remains but the type is added after the variable name - this is sometimes referred to as a *type hint*.

```
var i: Int = 1
var s: String = "String"
var n: Int = "error"  ①
```

- ① This assignment will fail with a compile error `error: type mismatch: inferred type is String but Int was expected`

In Kotlin `var` has a friend with the `val` keyword. Variables declared with `val` are immutable and cannot be written after assignment. It's the equivalent of `final var` in Java - a highly recommended default for any variable that isn't expected to be reassigned. Kotlin nicely makes this safer setting just as concise as the mutable alternative.

```
var i = 1          ①
i = 2

val s = "String"
s = "boom"        ②
```

- ① Reassignment is allowed to `var` variables
- ② Compile error `error: val cannot be reassigned`

`var` and `val` are used consistently through Kotlin for variables and arguments. The theme of favoring immutability by default is also a key design factor in Kotlin that we'll see time and

again throughout the language.

Once we have variables it's natural to want to compare them. Kotlin has some fresh help to give us on equality that's worth knowing about.

9.2.3 Equality

A common mistake in many Java programs is something like the following code.

```
// Java
String s = retrieveString();           ①
if (s == "A value") {
    // ...
}
```

- ① Receives a string from somewhere. Note that string literals may actually be interned to the same object, giving a false sense of security in the incorrect comparison.
- ② Not reached even when same value but separate references

We quickly learn that `==` in Java compares *references*, not *values* and so this doesn't do what you'd expect from many other languages.

Kotlin eliminates this quirk and treats `==` as value equality on common types such as `String`. Effectively calls to `==` are equivalent to a null-safe call to `equals`. This optimizes for the more common programming case and avoids a huge cause of errors in Java programming.

```
// Kotlin
var s: String = retrieveString()
if (s == "A value") {                 ①
    // ...
}
```

- ① Will be reached if the value of `s` is "A value"

On rare occasions you may still have reason to compare references. When those cases arise, Kotlin's `==` (and its pair `!=`) give you the behavior Java used for `==` and `!=`.

Comparing variables is important, but you won't get far without needing to call other code or define your own subroutines.

9.2.4 Functions

While sometimes when speaking we use the terms "function" and "method" interchangeably, in fact Java only has methods - you can't define a reusable block of code outside the context of a class. While Kotlin supports all the object-oriented goodness of Java - we'll see it in an upcoming section - it also recognizes that sometimes you just want a function by itself.

Keeping to Kotlin's principle of conciseness, a minimal function definition looks like this:

```
fun doTheThing() {  
    println("Done!")  
}
```

- ➊ fun defines a function in Kotlin. We'll see it too when we get to defining methods attached to classes.

This looks a little different to the Java way of doing things, but is still fairly recognizable. Apart from the `fun` keyword to start the declaration, the biggest difference is the lack of return type. In Kotlin if your function doesn't return anything, rather than stating `void` explicitly you can say nothing, and the return type is treated as `Unit` (Kotlin's way of saying "no return value").

If we do want to return a value, we must state that directly.

```
fun doTheThing(): Boolean {  
    println("Done!")  
    return true  
}
```

- ➊ Declare that our function returns type `Boolean`
- ➋ Return our success

Functions aren't much use without taking arguments. Kotlin's syntax for that looks much like variable declaration.

```
fun doTheThing(value: Int): Boolean {  
    println("Done $value!")  
    return true  
}
```

- ➊ Our function now takes a single argument of type `Int`
- ➋ Arguments appear within the function like locally defined variables. Here we use one in Kotlin's handy string interpolation.

Even the simple function definition in Kotlin has a few tricks up its sleeve, though. Sometimes the order of arguments to a function can be unclear - particularly when types match between arguments.

```
fun coordinates(x: Int, y: Int) {  
    // ...  
}
```

When we call this function we have to remember the order of the arguments - `x` comes before `y` - or risk a bug. Kotlin solves this with *named arguments*.

```
fun coordinates(x: Int, y: Int) {
    // ...
}

coordinates(10, 20)      ①
coordinates(y = 20, x = 10) ②
```

- ① Normal positional call to the function
- ② This call has the same result despite reordering because we name the arguments.

NOTE

Named arguments can't be used when calling Java functions, or conversely when calling a Kotlin function from Java - the names of arguments aren't preserved in the bytecode to allow this. Also, altering argument names can be considered a breaking API change in Kotlin code, where in Java only changes to type, number, or ordering of arguments will cause trouble.

Sometimes it's not necessary to pass all arguments to a function - there are reasonable defaults. In Java we accommodate this with multiple method overrides taking varying sets of arguments. While this is supported in Kotlin too, there's a more direct approach for default values.

```
fun callOtherServices(value: Int, retry: Boolean = false) {
    // ...
}

callOtherServices(10)      ①
```

- ① `retry` in the function is `false` thanks to the default value

Instead of needing to provide two definitions of `callOtherServices` - one with a single argument, another with the two - we can keep all the related bits in one function without boilerplate.

Another common case that Kotlin provides syntax for is single line functions. These provide encapsulation and a name to a particular calculation or check. Kotlin supports this as part of its trend towards conciseness via an alternate declaration for such short functions.

```
fun checkCondition(n: Int) = n == 42
```

This format not only is shorter for the lack of curly braces in favor of the single `=`, but also because type inference. Return types may be excluded, and Kotlin will infer the type of the expression automatically.

This feature hints towards a deeper part of Kotlin's design, which is its support for *first-class functions*. Functions in Kotlin may be passed as arguments, stored in variables and properties,

and returned from other functions. While Kotlin isn't considered a functional programming language, support for first-class functions allows Kotlin to benefit from many common patterns in functional programming.

Assigning a function to a variable can take a few different forms, depending on the source of your function. If we've previously declared the function, as can use the `::` operator to reference it by name.

```
fun checkCondition(n: Int) = n == 42
val check = ::checkCondition
```

Kotlin also has *lambda expression* syntax to create an anonymous function on the fly.

```
val anotherCheck = { n: Int -> n > 100 }
```

Regardless how it was assigned, references to functions can be invoked as if the variable name were the function's own name. Alternatively, an `invoke` function is available if that's clearer.

```
println(check(42))          ①
println(anotherCheck.invoke(42)) ②
```

- ① Runs our previously `fun` declared `checkCondition` captured in the `check` variable and prints `true`
- ② Runs our lambda to see if we're greater than 100 and prints `false`

We're not limited to assigning functions to local variables. They may be passed to other functions as arguments like any other value. This is one of the key properties of languages with first-class functions.

As we've seen previously, Kotlin requires arguments to declare their type. Functions are no exception, and there's a specific syntax for expressing the type of functions.

```
fun callsAnother(funky: (Int) -> Unit) {
    funky(42)
}

callsAnother({ n: Int -> println("Got $n") }) ③
```

- ① `callsAnother` takes an argument which is a function taking an `Int` and returning nothing
- ② `callsAnother` invokes the function it is passed
- ③ We can invoke `callsAnother` passing a lambda whose function type matches

A function type is composed of two parts separated by an `-` its list of arguments in `()` and then the return type. The list of argument types can be empty, but the return type cannot be excluded. If the function you are passing doesn't return anything, its type must be specified as `Unit`.

NOTE

When a lambda takes a single argument and the type can be inferred, the identifier `it` can be used, excluding the need both for a specific name and for the `at` at the start of the lambda.

While function arguments must specify types they expect callers to pass, Kotlin does save some typing for the caller by applying type inference to lambdas. The following are all allowed forms of our earlier invocation of `callsAnother` with progressively less and less explicit typing.

```
callsAnother({ n: Int -> println("Got $n") })      ①
callsAnother({ n -> println("Got $n") })          ②
callsAnother({ println("Got $it") })                 ③
```

- ① Original invocation of `callsAnother` with our lambda types fully specified
- ② Kotlin can infer that `n` must be an `Int` because that's what `callsAnother` needs
- ③ The pattern of passing a single parameter to lambda is so common, Kotlin provides special support with an implicit `it` parameter

Kotlin has one more trick when passing lambdas as arguments. If a function call's last argument is a lambda, that lambda may appear outside after the parentheses. If the only argument to a function is a lambda, you don't even have to use parentheses at all! The following three calls are all identical.

```
callsAnother({ println("Got $it") })
callsAnother() { println("Got $it") }
callsAnother { println("Got $it") }
```

We'll dig into more details of functional programming with Kotlin in chapter 13, but let's look at how Kotlin puts all this functional goodness to use improving a point of frequent discontent in Java - collections.

9.2.5 Collections

Collections are one of the most common data structures in programs. Java's standard collection library, from the earliest versions, has provided a lot of power and flexibility. But constraints of both the language and backwards compatibility often end up with more verbose, ceremony-laden code - especially when compared to scripting languages like Python or functional languages like Haskell. Recent Java releases have improved the situation dramatically - see Chapter 1's coverage of Collections Factories and Appendix B for Streams - but difficulties from Java's original collection design remain with us today.

Naturally, Kotlin learned from these mistakes and from day one had a seamless collection experience. Standard functions have always existed in Kotlin for creating the most common types of collections - a feature which only arrived in Java with version 9.

```

val readOnlyList = listOf("a", "b", "c")          ①
val mutableListOf = mutableListOf("a", "b", "c")  ①

val readOnlyMap = mapOf("a" to 1, "b" to 2, "c" to 3) ②
val mutableMap = mutableMapOf("a" to 1, "b" to 2, "c" to 3) ②

val readOnlySet = setOf(0, 1, 2)                  ③
val mutableSet = mutableSetOf(1, 2, 3)              ③

```

- ① Create lists with inferred types `kotlin.collections.List<String>` and `kotlin.collections.MutableList<String>`
- ② Create maps with inferred types `kotlin.collections.Map<String, Int>` and `kotlin.collections.MutableMap<String, Int>`. Note the built-in syntax for defining maps with the `to` keyword
- ③ Create sets with inferred types `kotlin.collections.Set<String>` and `kotlin.collections.MutableSet<String>`

The default functions return read-only copies of their collections - a great default choice for performance and correctness. You have to explicitly ask for the `mutable` flavor to get a collection with the interfaces for modification. Kotlin again aims to protect you from whole classes of bugs by nudging your code toward immutability as the easier, shorter option.

You may have noted the inferred types for these collections differ from the standard Java counterparts - while similarly named, they live in the `kotlin.collections` package. Kotlin defines its own hierarchy of collection interfaces but under the covers reuse the implementations from the JDK. This allows cleaner APIs with the `kotlin.collections` interfaces, while preserving the ability to pass our collections across to Java code, since the implementations also support the `java.util` collection interfaces.

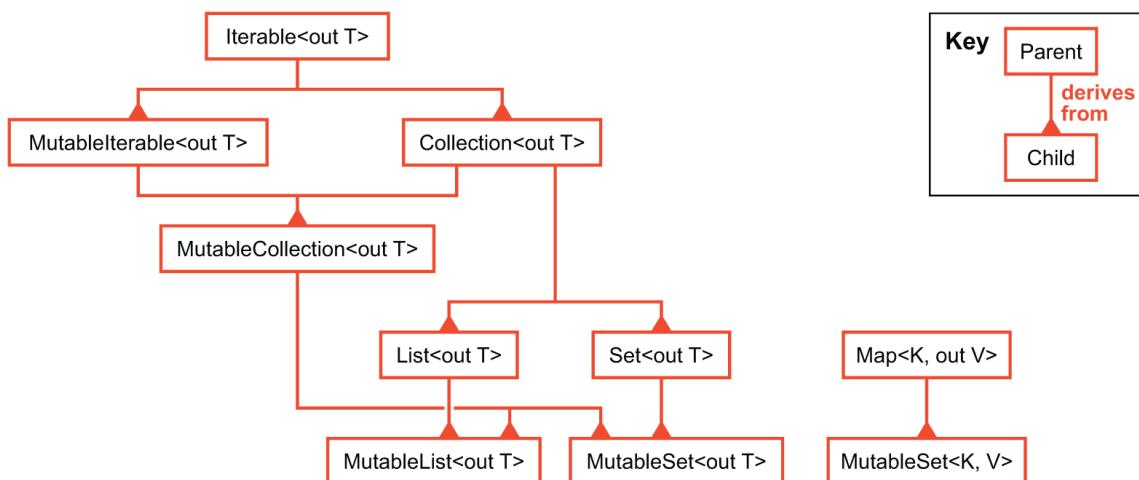


Figure 9.1 Kotlin collection hierarchy

These collections participate in all the standard Java interfaces and patterns. You can iterate over

them with `for ... in`.

```
val l = listOf("a", "b", "c")

for (s in l) {
    println(s)
}
```

However, the `for` loop to iterate over a collection is the only operation directly in the language. A lot of other work on collections happen over and over, and Kotlin's collections have a ton of features using the first-class functions we saw in the prior section. These features almost always return a new collection rather than mutating the collection they are called against. You may have encountered this style of collection code in Java since the release of lambdas and streams which share many common ideas.

Often we take a value in one collection and transform each element to a different value based on some calculation. `map` does exactly this using the function we pass in.

```
val l = listOf("a", "b", "c")
val result = l.map { it.toUpperCase() } ①
```

- ① `result` contains a list of "A", "B", "C"

Another common operation is to remove certain values from a collection before doing further processing. `filter` expects a lambda that returns a `Boolean`. This lambda is called a *predicate*, and `filter` calls the predicate repeatedly to decide which elements to return in a new collection.

```
val l = listOf("a", "b", "c")
val result = l.filter { it != "b" } ①
```

- ① `result` contains a list of "a", "c"

If you only care whether a collection fulfills certain conditions but you don't need the elements, the `all`, `any`, and `none` functions are just what you need. These avoid copying data and will return early where possible (i.e. after the first `false` for `all()`)

```
val l = listOf("a", "b", "c")
val all = l.all { it.length == 1 } ①
val any = l.any { it.length == 2 } ②
val none = l.none { it == "a" } ③
```

- ① `all == true`
- ② `any == false`
- ③ `none == false`

You can build a map from a list with the functions `associateWith` and `associateBy`. `associateWith` expects the collection element to be the key in the resulting map. `associateBy`

assumes instead the collection element is the value in the map. If duplicates are encountered with either of these functions, the last calculated value wins.

```
val l = listOf("!", "-", "--", "---")
val resultWith = l.associateWith { it.length }      ①
val resultBy   = l.associateBy   { it.length }      ②
```

- ① `resultWith` contains `mapOf("!" to 1, "-" to 1, "--" to 2, "---" to 3)`
- ② `resultBy` contains `mapOf(1 to "-", 2 to "--", 3 to "---")`

This just scratches the surface of the rich set of functions in Kotlin for working with collections. These functions can be chained together to allow expressive, concise descriptions of operations over a collection. [The documentation](#) is excellent with examples that walk through other topics like grouping, sorting, aggregating, and copying.

Kotlin's focus on maintaining a tight flow between bits of code carries to other fundamental features too. A preference for expressions over statements is another way Kotlin smooths the edges in your code.

9.2.6 Express yourself

Among the first structures we encounter when learning to program is `if`. In Java `if` is a statement used to control the flow of execution through your program. Kotlin also uses `if` for this purpose, but rather than a *statement* which is just executed, `if` is an *expression* which returns a value.

```
val iffy = if (checkCondition()) {      ①
    "sure"
} else {
    "nope"                         ②
}                                         ③
```

- ① variable `iffy` will receive a value depending on which branch is taken
- ② If `checkCondition()` is `true`, "sure" will be assigned to `iffy`
- ③ If `checkCondition()` is `false`, "nope" will be assigned to `iffy`

Like with any other variable assignment, Kotlin allows us to infer the type. In this case the final line of each branch is accounted for in determining the type.

`if` expressions are powerful enough that Kotlin actually dropped a feature which Java inherited from C - the ternary `condition ? "sure" : "nope"` operator. While the ternary shortens code, it also has a reputation for compressing to the point of losing readability. While slightly more characters, Kotlin's version is more readable in many cases and naturally converts to a multiline `if` should the logic grow further.

```
val myTurn = if (condition) "sure" else "nope"
```

In Chapter 1 we discussed the introduction of *switch expressions* to Java. This is another case where Kotlin's design preceded similar enhancements in Java. Kotlin doesn't use the traditional C-style `switch` syntax at all, but supports a powerful alternative with the keyword `when`.

```
val w = when (x) {
    1 -> "one"           ①
    2 -> "two"           ②
    else -> "lots"        ③
}
```

- ① If value `x` is 1, then we'll assign "one" to `w`
- ② If value `x` is 2, then we'll assign "two" to `w`
- ③ If value `x` is anything else, then we'll assign "lots" to `w`

`when` supports a number of other very useful forms. With the `in` keyword, you can check for membership in a collection.

```
val valid = listOf(1, 2, 3)
val invalid = listOf(4, 5, 6)

val w = when (x) {
    in valid    -> "valid"          ①
    in invalid  -> "invalid"        ①
    else         -> "unknown"
}
```

- ① Checks whether `x` is in each collection, equivalent of calling `valid.contains(x)` and `invalid.contains(x)`

Kotlin also has language level support for numeric ranges which plays nicely with `when` and `in` too.

```
val w = when (x) {
    in 1..3 -> "valid"          ①
    in 4..6 -> "invalid"        ①
    else       -> "unknown"
}
```

- ① The `..` syntax defines an inclusive range, so this code is equivalent to the prior list based example.

It's worth noting that the left hand condition of `when` can be any valid expression as long the type matches what's required. For example, function calls can be used, a nice trick for clarifying complex conditions.

```

fun theBest() = 1
fun okValues() = listOf(2, 3)

val message = when (incoming) {
    theBest()      -> "best!"          ①
    in okValues() -> "ok!"            ②
    else           -> "nope"
}

```

- ① Because the return value from `theBest` is used directly, it must return an `Int` to be compared to `x`
- ② Because the return value from `okValues` is used with an `in`, it must be a collection

A final point on `when`, if you aren't already convinced of its superpowers, is safety. All of our examples provide an `else` case. Removing any of those causes a compilation error complaining we haven't handled all the cases.

```
error: 'when' expression must be exhaustive, add necessary 'else' branch
```

Kotlin has one other trick of replacing a statement construct in Java with an expression - error handling with `try catch`.

```

val message = try {
    dangerousCall()          ①
    "fine"                  ②
} catch (e: Exception) {
    "oops"                  ③
}

```

- ① Function call which might fail
- ② `message` will be assigned "`fine`" if we get past the dangerous call
- ③ `message` will be assigned "`oops`" if our dangerous call threw an exception

This avoids the awkward construct of declaring a variable outside the `try catch` which all paths inside must remember to set properly. Not only is this shorter to write, but it's safer too since the compiler can guarantee the assignment is valid.

While both Kotlin and Java have adopted aspects of functional programming, they are at heart object oriented languages. Next we'll examine defining classes and objects in Kotlin.

9.3 A different view of classes and objects

Kotlin's classes provide very similar functionality to Java, starting with the keyword `class`. But, just as we've seen elsewhere, the code is different with an emphasis on conciseness and convenience.

For starters, Kotlin doesn't use the `new` keyword for creating instances of a class. Instead its

syntax is more akin to invoking a function using the class name.

```
val person = Person()
```

Kotlin doesn't really have fields in the same way that Java does. Instead, our friends `val` and `var` show up when declaring *properties* within the class.

```
import java.time.LocalDate

class Person {
    val birthdate = LocalDate.of(1996, 1, 23)      ①
    var name = "Name Here"                         ②
}
```

- ① Read-only property `birthdate`
- ② Mutable property `name`

NOTE

As we saw in Chapter 4, fields are present at JVM level, so Kotlin's properties are actually converted into field access in the bytecode. However, at the language level it is better to think in terms of properties.

A major source of boilerplate in Java classes is getter and setter methods for fields. Kotlin addresses this by providing accessors for properties automatically by default. To take it a step further, Kotlin also allows us to use those accessor methods as if we were accessing a field in Java:

```
println("Hi ${person.name}. You were born on ${person.birthdate}")      ①
person.name = "Somebody Else"                                              ②
// person.birthdate = LocalDate.of(2000, 1, 1)                            ③
```

- ① Prints Hi Name Here. You were born on 1996-01-23
- ② `var` properties also get setters which can be used with `=`
- ③ `val` properties cannot be set

A big topic in class designing is visibility of state data - especially as it relates to *encapsulation*. Kotlin takes a slightly controversial move of defaulting visibility to `public`, which differs from Java's *package-protected* default. While freely exposing all properties isn't considered good practice, Kotlin's designers found that `public` had to be stated far more often in real code, so choosing it as the default has a major slimming effect.

Kotlin supports four levels of visibility, most of which align with their Java counterparts.

- *private* - only visible within the current class or file for top-level functions
- *protected* - visible within the class and child classes
- *internal* - visible to the set of code you compile together
- *public* - visible to everyone

For example, if we wanted to make `birthdate` private it would look like this:

```
class Person {
    private val birthdate = LocalDate.of(1996, 1, 23)
    var name = "Name Here"
}
```

Because Kotlin exposes only properties, not fields, to the programmer, it opens the possibility for *delegated properties*. When follow with the `by` keyword, a property can provide a custom implementation of its getting and setting behavior. This will show up in a number of advanced techniques in later chapters.

Several useful delegates come with the standard library. For instance, it's not uncommon when debugging to want to know when a value was changed. `Delegates.observable` provides just such a hook.

```
import kotlin.properties.Delegates

class Person {
    var name: String by Delegates.observable("Name Here") {
        prop, old, new -> println("Name changed from $old to $new")
    }
}
```

The lambda we pass to `Delegates.observable` will be invoked after the backing property value has been changed. A handle to the property itself, along with the old and new values, are passed into the lambda for us to work with. Here we simply print out what changed.

Like Java, Kotlin supports constructors for creating instances of our classes - and in fact Kotlin actually has a few different forms for construction. The first of these is declaring a *primary constructor* at the very top with your class name. Kotlin uses this as an alternative location where you can specify your properties as well.

```
class Person(
    val birthdate: LocalDate,
    var name: String) {  
    ①  
}  
  
val person = Person(LocalDate.of(1996, 1, 23),
    "Somebody Else")  
    ②
```

- ① `val` and `var` in the primary constructor create properties so we don't need to declare them later
- ② Because we didn't provide defaults, parameters must be passed at construction

If you need visibility modifiers or annotations on a constructor, there is a longer syntax using the `constructor` keyword. For instance, if wanted to hide our constructor from the world we can do it like this.

```
class Person private constructor(
    val birthdate: LocalDate,
    var name: String) {
}
```

If we want to run other logic during all object construction, Kotlin uses the `init` keyword.

```
class Person(
    val birthdate: LocalDate,
    var name: String) {

    init {
        if (birthdate.year < 2000) {      ①
            println("So last century")
        }
    }
}
```

- ① `init` runs after we've assigned the properties from the constructor so we can access them in our code

A class may have multiple `init` blocks. They are run in the order they're defined in the class. Properties defined in the class body are only accessible to `init` blocks after the definition.

```
class Person(
    val birthdate: LocalDate,
    var name: String) {

    init {
        // println(nameParts)          ①
    }

    val nameParts: List<String> = name.split(" ")

    init {
        println(nameParts)          ②
    }
}
```

- ① Fails to compile with error: variable 'nameParts' must be initialized
- ② Works as expected and prints out list

If we need additional constructors, we define them in the class body with the `constructor` keyword. These are referred to as *secondary constructors*.

```
class Person(
    val birthdate: LocalDate,
    var name: String) {

    constructor(name: String)
        : this(LocalDate.of(0, 1, 1), name) {      ①
    }
}
```

- ① When a class has a primary constructor, secondary constructors must call it (directly or through other secondary constructors) via `this`

NOTE

Many cases in Java where multiple constructor overrides exist for supplying defaults can be handled in Kotlin with default argument values instead.

While a class with only fields can be of use, most of the time our classes have other functionality too. This uses the familiar function syntax we've already seen earlier in the chapter.

```
class Person(
    val birthdate: LocalDate,
    var name: String) {

    fun isBirthday(): Boolean {
        val today = LocalDateTime.now().toLocalDate()
        return today == birthdate
    }
}
```

As mentioned before, functions in Kotlin default to `public` visibility. If we want to hide a function precede it with the desired access modifier.

```
class Person(
    val birthdate: LocalDate,
    var name: String) {

    fun isBirthday(): Boolean {①
        return today() == birthdate
    }

    private fun today(): LocalDate {②
        return LocalDateTime.now().toLocalDate()
    }
}
```

- ① `isBirthday` is available for anyone who can see the `Person` class
- ② `today` is only available within the `Person` class

Another key part of object oriented programming is *inheritance*. Kotlin doesn't have the `extends` keyword, but instead expresses inheritance via the familiar `:` syntax we've seen in type declarations previously.

```
class Child(birthdate: LocalDate, name: String) {①
    : Person(birthdate, name) {②
}
```

- ① Parameters to the `Child` constructor. Note these are *not* marked `val` and `var`, so they don't collide with the parent properties but are available as local variables to pass to the superclass constructor
- ② Calling to the superclass constructor

This requires one other change to the `Parent` class. To encourage only subclassing where we intend and plan for it, Kotlin classes are *closed* by default. If a class may be subclassed, it must

use the `open` keyword. This is the inverse of the situation in Java, where classes are open by default and use `final` to indicate that they may *not* be subclassed.

```
open class Person(①
    val birthdate: LocalDate,
    var name: String) {
    //...
}
```

- ① `open` precedes the `class` keyword along with visibility modifiers

The same closed by default principle applies to methods. The parent class must declare method to be overridden `open`, and overrides must be marked with `override` in the child.

```
open class Person(
    val birthdate: LocalDate,
    var name: String) {

    open fun isBirthday(): Boolean {①
        return today() == birthdate
    }

    private fun today(): LocalDate {
        return LocalDateTime.now().toLocalDate()
    }
}

class Child(birthdate: LocalDate, name: String)
    : Person(birthdate, name) {

    override fun isBirthday(): Boolean {②
        val itsToday = super.isBirthday()③
        if (itsToday) {
            println("YIPPEE!!")
        }
        return itsToday
    }
}
```

- ① `Person` class declares that the `isBirthday` function can be overridden in subclasses.
- ② `Child` class must explicitly mark its method as an `override`
- ③ `Child` can call the parent implementation of the `isBirthday` with `super`

Like Java, Kotlin only allows a single base class, but classes may inherit multiple interfaces. Kotlin's interfaces allow default implementations of functions much like Java has since version 8.

```

interface Greetable {
    fun greet(): String
}

open class Person constructor(
    val birthdate: LocalDate,
    var name: String): Greetable { ①

    override fun greet(): String { ②
        return "Hello there"
    }
}

```

- ① Define our interface with a function to return a greeting
- ② Person declares that it implements Greetable
- ③ Interface functions are open, so their implementation must specify `override`

In typical Kotlin style, implementing an interface uses a concise form that looks a lot like what we already use for extending a base class. No more remembering whether to `extends` or `implements` like in Java.

9.3.1 Data classes

While these basic constructs allow us to make rich object models in Kotlin, sometimes you just want a container to pass data around. Doing this correctly in Java requires lots of boilerplate - you need the constructor, declarations for fields, getters and setters, implementation of equality and hashing. IDE generators and some libraries can help, but there's only so far Java can be pushed.

Kotlin already makes the property side of this seamless, but the equality issues remain with a standard classes - the default `equals` and `hashCode` implementations are based on object references, rather than the values of its properties.

Kotlin provides an alternative, though, with *data classes*. We can declare a type as a `data class` and Kotlin will create the equality functions we'd want.

```

class PlainPoint(val x: Int, val y: Int)

val p1 = PlainPoint(1, 1)
val p2 = PlainPoint(1, 1)

println(p1 == p2) ①

data class DataPoint(val x: Int, val y: Int)

val pd1 = DataPoint(1, 1)
val pd2 = DataPoint(1, 1)

println(pd1 == pd2) ②

```

- ① Default `equals` compares reference equality, so this prints `false`
- ② With Kotlin's `data class` implementation, this prints `true`

Data classes must have a primary constructors with at least one `val` or `var`. They can't be `open`, as it's impossible at compile-time for Kotlin to correctly generate the equality functions if child classes might exist for the type. Data classes also aren't allowed as inner classes. Apart from these and a handful of more exotic constraints, though, they are normal classes which you can implement functions or interfaces on to your heart's content.

NOTE

We met Java's new `record` capability in Chapter 3 - and Kotlin data classes are very similar to Java records in some ways.

A last feature that folks coming from Java might look for in classes is the declaring a function that belongs not to instances, but to the class a whole. Kotlin chose not support `static` though - functions are either free-floating, or they are members of a type.

However, the convenience of associating functions with class can't be denied, and Kotlin provides similar functionality via its `companion object`. This syntax declares a singleton object which lives within the class. `object` declarations in Kotlin are full objects with typical properties and functions. This avoids weird edges that `static` methods suffer from in Java (i.e. testing difficulties), while retaining the convenience of keeping functionality associated to the class.

A common use-case for these functions is factory methods, where you want to keep the constructors of your objects private but allow controlled creation by more specifically named methods.

```
class ShyObject private constructor(val name: String) { ①

    companion object {
        fun create(name: String): ShyObject { ②
            return ShyObject(name)
        }
    }
}

ShyObject.create("The Thing") ③
```

- ① `ShyObject` declares its constructor `private` so no one outside the class can use it
- ② Our factory method inside the `companion object` is part of the `ShyObject` class, so it can access the private constructor.
- ③ Outside our class, functions on ``ShyObject`'s companion are available directly via its class name.

As a pragmatic alternative to Java, Kotlin brings a lot of convenience and boilerplate reduction to the table. But it doesn't stop there, as we'll see in the next section.

9.4 Safety

Kotlin is built on top of the JVM, and so it has no choice but to live within some design constraints that really come from the design of the virtual machine. For example, the JVM specification defines `null` as a value that can be assigned to any variable of reference type.

Despite these issues, the Kotlin language seeks to address some common code safety concerns in order to try and minimize the inherited pain and suffering. This manifests itself by elevating a number of Java code patterns to language features to make your code safer by default.

9.4.1 Null safety

Among the most common Java exceptions is the `NullPointerException`. This happens when we try to access a variable or field that should have contained an object but instead was `null`. Nulls have been referred to by Tony Hoare, the original creator of the Quicksort algorithm, as his "[billion dollar mistake](#)" given his role in introducing the null reference in ALGOL.

Over it's history, Java has developed several different approaches to provide protection against nulls. The `Optional` type lets you always have a concrete object, while still indicating a "missing" value without resorting to `null`. The `@NotNull` and `@Nullable` annotations, supported by many different validation and serialization frameworks, can ensure that values aren't unexpectedly null at key points in our applications.

As you might expect, Kotlin has taken these common patterns and baked them right into the language itself. Let's revisit our earlier with assigning variables in this chapter. How do they behave when combined with nulls?

```
val i: Int = null      ①
val s: String = null  ①
```

- ① Trying to assign `null` to these types will fail to compile

Both assignments give a compilation error, `error: null can not be a value of a non-null type Int`. Although those `Int` and `String` type declarations look like Java's, they in fact disallow `null` values.

NOTE

Kotlin has made nullability part of its type system. The Kotlin type `String` is not actually the same as the Java type `String` that allows `null`.

For a Kotlin variable to allow `null`, we must state it explicitly by adding a suffix of `?` to the type:

```
val i: Int? = null      ①
val s: String? = null  ①
```

- ① Changing our types to `Int?` and `String?` will tell Kotlin to allow nulls

NOTE

Whenever possible declare your variables and arguments with non-null types.
You can rest assured that Kotlin is protecting you from those
NullPointerException headaches.

We can't always avoid nulls, though. Perhaps we're interacting with Java code or our classes weren't designed with null-safety in mind. Even once we've dipped our toes into the dangers of nullability, Kotlin still does its best to inform us of the risks.

```
val s: String? = null      ①
println(s.length)          ②
```

- ① Create a nullable variable
- ② Attempt to access a property on that variable

Kotlin recognizes the dereference in calling `s.length` is potentially unsafe and refuses to compile with error: `only safe (?) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?`.

The first option Kotlin suggests is to correct this to the safe operator `?..`. This operator examines the object its being applied to. If the object is `null` it returns `null` instead of making the further function call.

```
val s: String? = null      ①
println(s?.length)
```

- ① `?..` results in printing the value `null`

The safe operator returns early, so it works fine even with a nested chain of calls. Any point along the way in our following example can safely return a `null` and the whole expression will just turn to `null`.

```
data class Node(val parent: Node?, val value: String) ①
val node = getNode()
node.parent?.parent?.parent
```

- ① data class that allows for an optional parent node
- ② Retrieve a `Node` from somewhere
- ③ See if `node` has a great-grandparent node

`?..` potentially hides data issues though. If we got a `null` back on our great-grandparent check above, we can't assure which level of our hierarchy it came from without further inspection.

The second option for our compilation failure (`error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?.`) is using `!!` on the variable. This operator forces Kotlin to see if the object is `null` or not, and will raise the familiar `NullPointerException` if the value is `null`.

```
val s: String? = null
println(s!!.length)      ①
```

- ① throws a `NullPointerException`

Although it should be needed less often, we can still check whether a variable is `null` or not. In fact, Kotlin can often notice such checks and let us avoid further `?.` or `!!.`

```
val s: String? = null

if (s != null) {          ①
    println(s.length)     ②
}
```

- ① Checks for `null` in all cases
- ② Because we know from <1> that `s` isn't `null`, it can be safely referenced

What we've seen here is actually a deeper feature of Kotlin called *smart casting* which is worth looking at more closely in its own right.

9.4.2 Smart casting

Although good object-oriented design tries to avoid directly checking the type of objects, sometimes it's necessary. A data formats at the edges of our system may be loose about types (i.e. JSON) and often outside our control. At other types, we have plugins system that must dynamically probe for capabilities of an object.

Kotlin embraces this need and takes it a step further in how the compiler supports the common patterns. To start, Kotlin uses the `is` operator to check an object's type.

```
val s: Any = "Hello"
if (s is String) {          ①
    println(s.toUpperCase()) ②
}
```

- ① `Any` is the equivalent of Java's `Object` type - the base type for all objects
- ② Check whether `s` contains a `String` instance
- ③ Use the variable `s` as a `String`. `toUpperCase` wouldn't be available if the compiler treated it as type `Any` still within the branch.

If you're familiar with Java's `instanceof` construct, this code appears to miss a crucial step - we

look to see whether `s` is a `String`, but then we don't cast it before treating it as a `String`. Fortunately, Kotlin has us covered. Within the `if` block where the compiler can ensure we have a `String`, we may use `s` as a `String` without explicitly casting. This is known as `smart casting`.

NOTE

Java has a new feature called pattern matching that is being slowly rolled out as part of Project Amber. The first piece of it applies to `instanceof` and provides some of the same benefits as smart casting. We will discuss pattern matching in more detail in Chapter 17.

Kotlin's smart casting functionality is allowed within an `if` conditional as well.

```
val s: Any = "Hello"
if (s is String && s.toUpperCase() == "HELLO") {      ①
    println("Got something")
}
```

- ① Kotlin can ensure the type from our check on the left hand side of the `&&` so it can safely upper-case without casting.

There are constraints around where this smart casting can kick in. In particular it won't work with `var` properties on a class because values may mutate between its check and the following code, compromising safety.

Even if Kotlin can't do it directly, we may still cast to the type we expect - it's just a little less convenient.

```
if (s is String) {          ①
    val cast = s as String  ②
    println(cast.toUpperCase())
}
```

- ① Assumes `s` is defined in a way that we can't smart cast
- ② `as` casts to the expected type

When we use `as`, we're back to the same spot as in Java when we cast. We'll see `ClassCastException` if the types aren't actually compatible. Kotlin does provide an alternative if we would prefer allowing nullability into the picture instead of exceptions.

```
val cast: String? = s as? String      ①
if (cast != null) {                   ②
    println(cast.toUpperCase())
}
```

- ① `as?` attempts the cast but won't throw. Note also that the resulting type is `String?` not `String`
- ② If `s` couldn't be cast, the variable will be `null` instead

A lot of Kotlin's power comes from taking a fresh look at the common, practical coding that Java developers have been doing for years. One area where the language provides more than just polish and protection, though, is in concurrency. Kotlin provides a technique called *coroutines* which can be thought of as an alternative to the classical threading approaches that are most widely used in Java.

9.5 Concurrency

As we discussed in Chapter 5, since its very first versions the JVM has supported the `Thread` class as a model of operating system managed threads. The thread model is well understood, but it comes with many problems.

NOTE

Whilst threads are so deeply embedded in the Java language and ecosystem that it would be almost impossible to remove them, moving to a new, non-Java language allows us to potentially re-imagine the concurrency primitives that the language might use.

Although Kotlin as a JVM language still exposes threads, it also introduces another construct called *coroutines*. At the simplest level, a coroutine can be thought of as a lighter-weight thread. These are implemented and scheduled within the runtime instead of at the operating system level, making them much less resource intensive. Spinning up thousands of coroutines isn't a problem at all, where similar counts of threads would grind a system to a halt.

NOTE

We'll meet Java's take on coroutines in Chapter 17 when we discuss Project Loom

Part of Kotlin's support for coroutines is directly in the language (`suspend` functions) but to use coroutines practically requires an additional library, `kotlinx-coroutines-core`. We'll see a lot more about introducing these sorts of dependencies in Chapter 11, but for now the addition would look like this in Maven.

```
<dependency>
    <groupId>org.jetbrains.kotlinx</groupId>
    <artifactId>kotlinx-coroutines-core</artifactId>
    <version>1.3.9</version>
</dependency>
```

The equivalent in Kotlin-flavored Gradle is as follows.

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9")
}
```

In Java a thread is started by handing it an object that implements the `Runnable` interface. Coroutines in Kotlin also need a way to receive the code to run, but instead they use the

language's lambda syntax.

A coroutine is always started in a *scope* which controls how the coroutine will be scheduled and run. We'll start with the simplest options which is `GlobalScope`, a scope which exists for the entire duration of your application. `GlobalScope` has a `launch` function we call with a lambda to get ourselves started.

```
import kotlinx.coroutines.GlobalScope      ①
import kotlinx.coroutines.launch

fun main() {
    GlobalScope.launch {                ②
        println("Inside!")
    }
    println("Outside")               ③
}
```

- ① Imports for the coroutine function and object we'll use
- ② Start a new coroutine in the `GlobalScope` which lives as long as our program does
- ③ Back outside of our coroutine we'll print to see `main` still runs

When we run this example, most often you'll simply see it output the following:

```
Outside
```

Why isn't our coroutine working? We expect at some point to see `Inside` printed as well. Looking closer, though, we can spot the problem if we think about the sequence of events. `main` starts up our program. We then launch our coroutine to run asynchronously. Following that, we print our `Outside` message, and then the program is finished. When `main` is done, the program exits, regardless what coroutines may be waiting to run.

To get the result we wanted, we need to introduce a pause before the program finishes. This could be done with a loop or asking for user input at the console. We'll just use a `Thread.sleep(1000)` to give enough time for everything to settle.

```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.launch

fun main() {
    GlobalScope.launch {                ①
        println("Inside!")
    }
    println("Outside")               ②
    Thread.sleep(1000)
}
```

- ① Start our coroutine again
- ② Give the coroutine time to run

Now we'll see output with both messages - although the order is potentially non-deterministic

depending on how fast the coroutine starts up and what the main thread is doing.

At a high level this doesn't look much different from using threads to get similar concurrent execution of code. But the underlying implementation requires fewer operating system resources (each coroutine doesn't have its own execution stack and local storage) and allows safety for operations like cancelling a coroutine.

To see this in action, we can capture a handle to the coroutine with the return value of `launch`. This coroutine object presents a `cancel` function which we can call immediately if we want.

```
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

fun main() {
    val co = GlobalScope.launch {           ①
        delay(1000)                      ②
        println("Inside!")
    }
    co.cancel()                         ③
    println("Outside")                  ④
    Thread.sleep(2000)
}
```

- ① Capture the coroutine object returned by `launch`
- ② Inside of coroutines we can call `delay` to wait a period of time
- ③ Cancel the coroutine immediately
- ④ Wait as long as you like here, you'll never see the coroutine output

This code will safely stop the coroutine and only print `Outside`. This is a marked contrast to the `stop()` method on `java.lang.Thread` which was deprecated long ago due to being hopelessly unsafe - as we discussed in Chapter 5.

Why can coroutines accomplish this safely where threads can't? The key is the `delay` function. Its declaration is marked with a special modifier: `suspend`. Kotlin knows to treat suspend functions as safe spots in coroutine execution for operations like switching to another task or looking for cancellations.

This is known as *cooperative multitasking*, and it's only because the code inside our coroutine "cooperates" in its calls to suspend functions that it can be cancelled.

This cooperation gives benefits beyond just the ability to safely cancel. For example, Kotlin understands when one coroutine (parent) starts another coroutine (child). Cancelling the parent automatically cancels the child coroutines without additional management on our part.

```

import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.coroutineScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

fun main() {
    val co = GlobalScope.launch {           ①
        coroutineScope {                 ②
            delay(1000)
            println("First")
        }
        coroutineScope {                 ②
            delay(1000)
            println("Second")
        }
    }
    co.cancel()                         ③
    Thread.sleep(2000)                  ④
}

```

- ① Start our parent coroutine as before
- ② Start two child coroutines. `coroutineScope` associates those to the enclosing scope - in this case our global coroutine.
- ③ Cancel the parent coroutine
- ④ Again, we can wait here but we won't see any output

If you've seen the implementation necessary in Java to accomplish this sort of coordination, the value Kotlin brings here is pretty apparent.

Coroutines are a great example of how Kotlin uses its strength as a separate language with its own compiler to work with libraries to accomplish a lot of complex behavior cleanly. In fact, there's enough in coroutines that we'll be back for a deeper investigation in Chapter 14.

But no language lives in a vacuum, especially on the JVM. Kotlin has seen great success and uptake because of its strong focus on interoperating with the vast world of Java code out there.

9.6 Java interoperability

As we learned in Chapter 2, the class file is the center of the JVM's execution model. It should come as no surprise that the Kotlin compiler (`kotlinc`) produces class files much like `javac` does for Java.

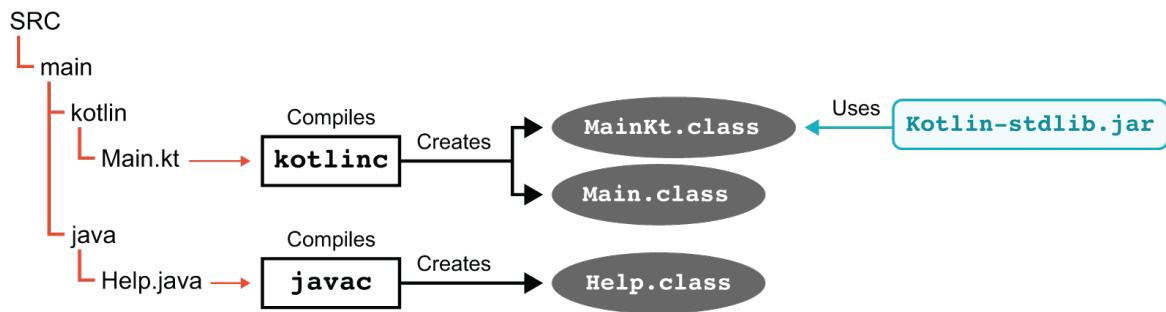


Figure 9.2 Kotlin and Java working side by side to generate class files

Basic class definitions end up looking similar between the languages, but when Kotlin provides a feature that isn't available in Java, we see more interesting differences in the generated class files. These are evidence of compiler fictions which we discussed in chapter 8.

An example is Kotlin's top-level functions outside of classes. This isn't even directly supported in the JVM class file format. Kotlin bridges this gap by generating a class with the `.kt` suffix named after its compilation file. Any top-level functions from within that file will appear in the `.kt` class.

NOTE

You can alter the name of the generated class with the `@file:JvmName("AlternateClassName")` annotation in your `.kt` file.

For example:

```
// Main.kt
package com.wellgrounded.kotlin

fun shout() {
    println("No classes in sight!")
}
```

①
②

- ① The filename by default influences the wrapping class name generated
- ② Users of the function will need to import from our package like usual

When compiled, this will produce a class file `MainKt.class` with our function in it. Since Java doesn't provide top-level functions itself, using the function from Java must go through that intermediate class.

```
// App.java
import com.wellgrounded.kotlin.MainKt;           ①

public class Main {
    public static void main(String[] args) {
        MainKt.shout();                         ②
    }
}
```

- ① Import the class Kotlin created to wrap the function
- ② Invoke the function through Java's static method syntax

Another key convenience in Kotlin is its built-in treatment of properties. A little `val` and `var`, and we never end up writing screens full of boilerplate getters and setters. Using Kotlin classes from Java reveals that at the lower level those methods have been there all along - Kotlin just wraps them up for our convenience.

```
// Person.kt
class Person(var name: String) { ①

// App.java
public class App {
    public static void main(String[] args) {
        Person p = new Person("Some Body"); ②
        System.out.println(p.getName()); ③

        p.setName("Somebody Else");
        System.out.println(p.getName());
    }
}
```

- ① Our property is `var`, so it is mutable
- ② Kotlin class still instantiated with `new` when used from Java
- ③ Accessing `Person.name` in Kotlin, is `Person.getName()` in Java
- ④ Accessing `Person.name = "..."` in Kotlin, is `Person.setName("...")` in Java. Note that this accessor is only available because the our `Person` class declares the `name` property a `var`, or mutable. If `name` was instead declared `val`, only the `getName()` accessor would be generated.

NOTE

This reveals that under the covers, Kotlin has been doing the standard pattern of creating a private field and wrapping access to the field. Kotlin lets us use the more natural property access form without the risks of exposing fields directly.

A number of other convenient features in Kotlin don't manifest in the resulting code when used from another JVM language. Named arguments are one example - Java just doesn't have a way to address an argument by name, so that nicety remains only in Kotlin code.

At a surface level, it might seem as though default values would suffer the same fate - after all calling from Java requires that you explicitly pass in all arguments to the function.

```
// Person.kt
class Person(var name: String) {
    fun greet(words: String = "Hi there") { ❶
        println(words)
    }
}

// App.java
public class App {
    public static void main(String[] args) {
        Person p = new Person("Some Body");
        // p.greet(); ❷
        p.greet("Howdy"); ❸
    }
}
```

- ❶ Standard Kotlin default value for argument words
- ❷ Can't invoke with default or we get a compile error reason: actual and formal argument lists differ in length
- ❸ We can pass our own value

However, there is an escape hatch that means that we don't have to abandon Kotlin's tidiness. The `@JvmOverloads` annotation tells Kotlin to explicitly generate the necessary variations of a function so calling it from other JVM languages looks the same.

```
// Person.kt
class Person(var name: String) {
    @JvmOverloads
    fun greet(words: String = "Hi there") { ❶
        println(words)
    }
}

// App.java
public class App {
    public static void main(String[] args) {
        Person p = new Person("Some Body");
        p.greet(); ❷
        p.greet("Howdy"); ❸
    }
}
```

- ❶ Annotate our Kotlin function and provide default as before
- ❷ Works fine and prints the default "Hi there"
- ❸ Works as before and prints our passed alternate greeting

Several other annotations allow for controlling how our Kotlin code manifests at the JVM level. One we've already seen in another context is `@JvmName`. This applies to functions as well as files to control the eventual naming outside Kotlin. `@JvmField` lets us avoid property wrappers and expose a bare field to the world if required.

Last but certainly not least is `@JvmStatic`. As we've seen earlier, Kotlin will wrap top-level functions in specially named classes which can be accessed as static methods in Java. There's one

prominent static method in all Java applications, even if you avoid statics otherwise - the `main` method that starts an application.

If we wanted to create an application in Kotlin, you can define its main method with `@JvmStatic` to avoid any weird naming needs on startup.

```
class App {  
    companion object {  
        @JvmStatic fun main(args: Array<String>) {  
            println("Hello from Kotlin")  
        }  
    }  
}
```

- ① App class we'd specify as our main class to start up
- ② `@JvmStatic` means this function will present as a static method on the containing class, not just on the companion

Making a change of language on a project is usually a huge step. Kotlin eases this burden, though, by leaning on standard patterns for multi-language projects on the JVM. Unsurprisingly, there's additional tooling as well if you're using IntelliJ. We'll look at the standard project layout in Chapter 11, but for now it's enough to know that projects typically embed the languages used into the directory layout like this.

```
src  
  main  
    java  
      JavaCode.kt  
    kotlin  
      KotlinCode.kt
```

This separation makes it easy for build tools to find what they need for all your code to coexist.

If you're using IntelliJ, the good folks at JetBrains have taken this a step further. Right-clicking on a Java file you'll find an item convert that single file directly to Kotlin. This makes it possible to start conversion on a system from whatever point makes the most sense - perhaps with tests, or a module which isn't deeply entangled in the remainder of an app.

The IDE will walk you through additional steps as necessary, but converting does take a little more than just switching some source files. Your build tools need to know about Kotlin to compile it alongside your existing code. Also, the Kotlin standard library `kotlin-stdlib` needs to be included in your project as a dependency. We'll see more about how to manage these sorts of dependencies in Chapter 11.

NOTE

While IntelliJ provides a Java to Kotlin translation, it does not go the other direction. Keep that source control handy as always when starting a big conversion.

The fact that Kotlin compiles to class files and provides much of its additional functionality through libraries means even including this new language in your project doesn't change that you're just running on the good old JVM.

9.7 Summary

Kotlin is a pragmatic, attractive alternative language on the JVM. It provides a huge number of conveniences and protections drawn from many years of production Java use. As a new language it made change that will likely never happen in Java due to backwards compatibility requirements.

Kotlin prizes its conciseness and safety. Familiar constructs in Java can almost always be written with less code in Kotlin. Kotlin also baked null safety into the language in ways reduce the risk of throwing a `NullPointerException` in production.

But the story doesn't stop with your application or library code. Kotlin script (`kts`) lets you write scripts that before may have sent you reaching for a dynamic language or shell. Even your build scripts can be written using Kotlin, as we'll see in detail in Chapter 11 when we discuss Gradle.

With the recent adoption of Kotlin as the recommended language for the Android platform, it's clear that Kotlin is a solid, well-supported option worth considering for those who hungering for a modern alternative to Java without giving up the power of the JVM.

10

Clojure: a different view of programming

This chapter covers

- Clojure's concept of identity and state
- The Clojure REPL
- Clojure syntax, data structures, and sequences
- Clojure interoperability with Java
- Clojure macros

Clojure is a very different style of language from Java and the other languages we've studied so far. Clojure is a JVM reboot of one of the oldest programming languages — Lisp. If you're not familiar with Lisp, don't worry. We'll teach you everything you need to know about the Lisp family of languages to get you started with Clojure.

In addition to its heritage of powerful programming techniques from classic Lisp, Clojure adds amazing cutting-edge technology that's very relevant to the modern Java developer. This combination makes Clojure a standout language on the JVM and an attractive choice for application development.

Particular examples of Clojure's new tech are its concurrency toolkits (which we will meet in Chapter 15) and data structures (which we will introduce here and expand on in Chapter 14).

For the avid reader who can't wait until later - let us just say this: The concurrency abstractions enable programmers to write much safer multithreaded code than when working in Java. These abstractions can be combined with Clojure's seq concept (a different take on collections and data structures) to provide a very powerful developer toolbox.

To access all of this power, some important language concepts are approached in a fundamentally different way from Java. This difference in approach makes Clojure interesting to

learn, and it will probably also change the way you think about programming.

NOTE

Learning Clojure will help make you a better programmer in any language. Functional programming matters.

We'll kick off with a discussion of Clojure's approach to state and variables. After some simple examples, we'll introduce the basic vocabulary of the language — the *special forms* that are the equivalent of keywords in languages like Java. A small number of these are used to build up the rest of the language.

We'll also delve into Clojure's syntax for data structures, loops, and functions. This will allow us to introduce sequences, which are one of Clojure's most powerful abstractions.

We'll conclude the chapter by looking at two very compelling features: tight Java integration and Clojure's amazing macro support (which is the key to Lisp's very flexible syntax). Later on in the book, we'll meet more Clojure goodness (as well as Kotlin and Java examples) when we talk about Advanced Functional Programming (Chapter 14) and Advanced Concurrency (Chapter 15).

10.1 Introducing Clojure

The basic unit of Lisp syntax consists of an expression to be evaluated. These are typically represented as zero or more symbols surrounded by brackets. If the evaluation succeeds without errors then the expression is called a *form*.

NOTE

Clojure is compiled, not interpreted, but the compiler is very simple. Also remember that Clojure is dynamically typed, so there won't be many type checking errors to help you - they will show up as runtime exceptions instead.

Simple examples of forms include:

```
0
(+ 3 4)
(list 42)
(quote (a b c))
```

The true core of the language only has a very few built-in forms (the special forms). They are the Clojure equivalent of Java keywords, but be aware that:

1. Clojure has a different meaning for the term "keyword", which we'll encounter later
2. Clojure (like all lisps) allows the creation of constructs that are indistinguishable from built-in syntax

When working with Clojure code it almost never matters whether the forms you're using are

special forms or library functions that are built up from them.

Let's get started with forms by looking at one of Clojure's most important conceptual differences from Java. This is the treatment of state, variables, and storage. As you can see in figure 10.1, Java (like Kotlin) has a model of memory and state that involves a variable being a "box" (really a memory location) with contents that can change over time.

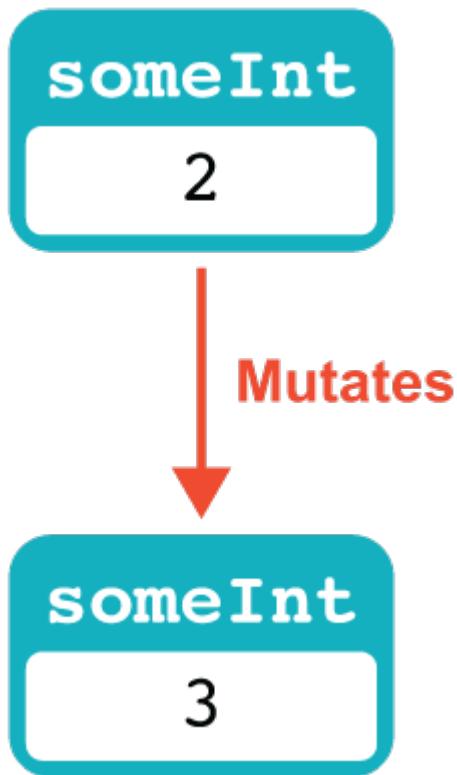


Figure 10.1 Imperative language memory use

Programming languages like Java are *mutable by default*, because we are trying to alter the program state, which in Java is made up of objects. Languages that follow this model are often called imperative languages, as we discussed in Chapter 8.

Clojure is a little bit different — the important concept is that of a *value*. Values can be numbers, strings, vectors, maps, sets, or a number of other things. Once created, values never alter. This is really important, so we'll say it again. *Once created, Clojure values can't be altered — they're immutable.*

NOTE

Immutability is a very common property of languages that are used for functional programming, because it allows mathematical reasoning techniques about the properties of functions (such as the same input always giving the same output) to be used.

This means that the imperative language model of a box that has contents that change isn't the way Clojure works. Figure 10.2 shows how Clojure deals with state and memory. It creates an association between a name and a value.

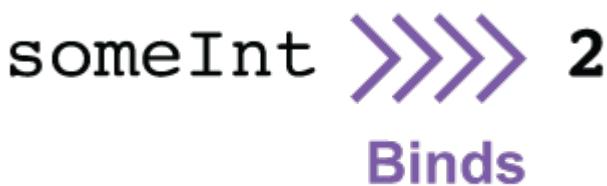


Figure 10.2 Clojure memory use

This is called *binding*, and it's done using the `def` special form. Let's meet the syntax for `(def)`:

```
(def <name> <value>)
```

Don't worry that the syntax looks a little weird — this is entirely normal for Lisp syntax, and you'll get used to it really quickly. For now you can pretend that the brackets are arranged slightly differently and that you're calling a method like this:

```
def(<name>, <value>)
```

Let's demonstrate `(def)` with a time-honored example that uses the Clojure interactive environment.

10.1.1 Hello World in Clojure

If you haven't already installed Clojure then you can do so on Mac, by running this command:

```
brew install clojure/tools/clojure
```

This will install the command line tools with `brew` from the `clojure/tools` tap. For other operating systems, instructions can be found on the clojure.org website.

NOTE

Windows support isn't so great for Clojure - for example `c1j` is still in an alpha state. Follow the instructions on the website carefully.

Once installed, you can use the `clj` command to start the Clojure interactive session. Or, if you built Clojure from source, change into the directory where you installed Clojure and run this command:

```
java -cp clojure.jar clojure.main
```

Either way, this brings up the user prompt for the Clojure read-evaluate-print loop (REPL). This is the interactive session, which is where you'll typically spend quite a lot of time when developing Clojure code. It looks like this:

```
$ clj
Clojure 1.10.1
user=>
```

The `user` part is the Clojure prompt for the session, which can be thought of as a bit like an advanced debugger or a command line. To exit the session (which will cause all the accumulated state in the session to be lost) the traditional Unix sequence `Ctrl-D` is used.

Let's write a "Hello World" program in Clojure:

```
user=> (def hello (fn [] "Hello world"))
#'user/hello

user=> (hello)
"Hello world"
user=>
```

In this code, you start off by binding the *identifier* `hello` to a value. `(def)` always binds identifiers (which Clojure calls *symbols*) to *values*. Behind the scenes, it will also create an object, called a *var*, that represents the binding (and the name of the symbol).

```
(def hello (fn [] "Hello world"))
-----  
|   |  
|   |           value  
| symbol  
|  
special form
```

What is the value you're binding `hello` to? It's the value:

```
(fn [] "Hello world")
```

This is a function, which is a genuine value (and so therefore immutable) in Clojure. It's a function that takes no arguments and returns the string "Hello world". The empty argument list is represented by the `[]`.

NOTE

In Clojure (but not in other Lisps), square brackets indicates a linear data structure called a vector - in this case a vector of function arguments.

After binding it, you execute it via `(hello)`. This causes the Clojure runtime to print the results of evaluating the function, which is "Hello world."

Remember that the round brackets mean "function evaluation" in Lisps, so the example basically consists of:

- Create a function and bind it to the symbol `hello`
- Call the function bound to the symbol `hello`

At this point, you should enter the Hello World example (if you haven't already), and see that it behaves as described. Once you've done that, we can explore a little further.

10.1.2 Getting started with the REPL

The REPL allows you to enter Clojure code and execute Clojure functions. It's an interactive environment, and the results of earlier evaluations are still around. This enables a type of programming called *exploratory programming*, which basically means that you can experiment with code. In many cases the right thing to do is to play around in the REPL, building up larger and larger functions once the building blocks are correct.

NOTE

Subdivision is a key technique in functional programming - breaking down a problem into smaller parts, until it becomes either soluble or amenable to a reusable pattern (which may already be in the standard library).

Let's look at a bit more Clojure syntax. One of the first things to point out is that the binding of a symbol to a value can be changed by another call to `def`, so let's see that in action in the REPL — we'll actually use a slight variant of `(def)` called `(defn)`:

```
user=> (hello)
"Hello world"

user=> (defn hello [] "Goodnight Moon")
#'user/hello

user=> (hello)
"Goodnight Moon"
```

Notice that the original binding for `hello` is still in play until you change it — this is a key feature of the REPL. There is still state, in terms of which symbols are bound to which values, and that state persists between lines the user enters.

The ability to change which value a symbol is bound to is Clojure's alternative to mutating state. Rather than allowing the contents of a storage location (or "memory box") to change over time,

Clojure allows a symbol to be bound to different immutable values at different points in time. Another way of saying this is that the var can point to different values during the lifetime of a program. An example can be seen in figure 10.3.



Figure 10.3 Clojure bindings changing over time

NOTE

This distinction between mutable state and different bindings at different times is subtle, but it's a very important concept to grasp. Remember, mutable state means the contents of the box change, whereas rebinding means pointing at different boxes at different points in time.

This is in some ways similar to the Java concept of `final` references. In Java, if we say `final int` then this means that the contents of the storage location cannot change. As ints are stored as bit patterns this means that the value of the int cannot change.

However, if we say `final AtomicInteger` then the contents of the storage location once again cannot change. This case is different, though, because a variable containing an atomic integer actually holds an object reference.

The atomic integer object stored in the heap can change the value it stores (whereas an `Integer` cannot) - and this is true whether or not the reference to the object is `final` or not.

We've also slipped in another Clojure concept in the last code snippet—the `(defn)` "define function" *macro*. Macros are one of the key concepts of Lisp-like languages. The central idea is that there should be as little distinction between built-in constructs and ordinary code as possible.

NOTE

Macros allow you to create forms that behave like built-in syntax. The creation of macros is an advanced topic, but mastering their creation will allow you to produce incredibly powerful tools.

This means that the true language primitives of the system (the special forms) can be used to build up the core of the language in such a way that you don't really notice the difference between the two.

NOTE

The `(defn)` macro is an example of this. It's just a slightly easier way to bind a function value to a symbol (and create a suitable var, of course) - it's not a special form, but instead is a macro built up from the special forms `(def)` and `(fn)`.

We will introduce macros properly at the end of this chapter.

10.1.3 Making a mistake

What happens if you make a mistake? Suppose you're trying to declare a function but accidentally just `def` a value instead:

```
user=> (def hello "Goodnight Moon")
#'user/hello

user=> (hello)
Execution error (ClassCastException) at user/eval137 (REPL:1).
class java.lang.String cannot be cast to class clojure.lang.IFn
(java.lang.String is in module java.base of loader 'bootstrap';
clojure.lang.IFn is in unnamed module of loader 'app')
```

There's a couple of things to notice here - first off that the error is a runtime exception. This means that the form `(hello)` compiled fine - it just failed at run time. In terms of the equivalent code in Java , it looks a bit like this (we've simplified things somewhat to make it easier to understand for folks who are new to Clojure or language implementation):

```
// (def hello "Goodnight Moon")
var helloSym = Symbol.of("user", "hello");
var hello = Var.of(helloSym, "Goodnight Moon");

// Or just
// var hello = Var.of(Symbol.of("user", "hello"), "Goodnight Moon");

// #'user/hello

// (hello)
hello.invoke();

// ClassCastException
```

where `Symbol`, `Var` are classes in the package `clojure.lang` that provides the core of the

Clojure runtime. They look similar to these basic implementations - which we have simplified:

```

public class Symbol {
    private final String ns;
    private final String name;

    private Symbol(String ns, String name) {
        this.ns = ns;
        this.name = name;
    }
    // toString() etc
}

public class Var implements IFn {
    private volatile Object root;
    public final Symbol sym;
    public final Namespace ns;

    private Var(Symbol sym, Namespace ns, Object root) {
        this.sym = sym;
        this.ns = ns;
        this.root = root;
    }

    public static Var of(Symbol sym, Object root){
        return new Var(sym, Namespace.of(sym), root);
    }

    static public class Unbound implements IFn {
        final public Var v;
        public Unbound(Var v){
            this.v = v;
        }

        @Override
        public String toString(){
            return "Unbound: " + v;
        }
    }
}

public synchronized void bindRoot(Object root) {
    this.root = root;
}

public synchronized void unBindRoot(Object root) {
    this.root = new Unbound(this);
}

@Override
public Object invoke() {
    return ((IFn)root).invoke();
}

@Override
public Object invoke(Object o1) {
    return ((IFn)root).invoke(o1);
}

@Override
public Object invoke(Object o1, Object o2) {
    return ((IFn)root).invoke(o1, o2);
}

@Override
public Object invoke(Object o1, Object o2, Object o3) {
    return ((IFn)root).invoke(o1, o2, o3);
}
// ...
}

```

The all-important interface `IFn` looks a bit like this:

```
public interface IFn {
    default Object invoke() {
        return throwArity();
    }
    default Object invoke(Object o1) {
        return throwArity();
    }
    default Object invoke(Object o1, Object o2) {
        return throwArity();
    }
    default Object invoke(Object o1, Object o2, Object o3) {
        return throwArity();
    }

    // ... many others including eventually a variadic form

    default Object throwArity(){
        throw new IllegalArgumentException("Wrong number of args passed: "
            + toString());
    }
}
```

`IFn` is the key to how Clojure forms work - the first element in a form is taken to be a function, or the name of a function, to be invoked. The remaining elements are the arguments to the function - and the `invoke()` method with the appropriate number of arguments (arity) is called.

If a Clojure var is not bound to a value that implements `IFn` then a `ClassCastException` is thrown at runtime. If the value is an `IFn` but the form tries to invoke it with the wrong number of arguments, then an `IllegalArgumentException` is thrown (it's actually a subtype called an `ArityException`).

NOTE

Remember that Clojure is dynamically typed - as you can see in several places e.g. all the arguments and return types of the methods in `IFn` are `Object` and that `IFn` is not a Java-style `@FunctionalInterface` but instead has multiple methods defined on it to handle many different arities.

This peek under the hood should help clarify both a little of Clojure's syntax and how it all fits together. However, we still have some broken code to fix - but fortunately it's not too hard!

All that's happened is that you've got your `hello` identifier bound to something that isn't a function so can't be called. In the REPL, you can fix this by simply rebinding it:

```
user=> (defn hello [] (println "Dydh da an Nor")) ; "Hello World" in Cornish
#'user/hello

user=> (hello)
Dydh da an Nor
nil
```

As you might guess from the preceding snippet, the semicolon `;` character means that everything

to the end of the line is a comment, and `(println)` is the function that prints a string. Notice that `(println)`, like all functions, returns a value, which is echoed back to the REPL at the end of the function's execution.

Clojure does not have statements like Java - only expressions. This means that all functions must return a value. If there is no value to return then `nil` is used, which is basically the Clojure equivalent of Java's `null`. Functions that would be `void` in Java will return `nil` in Clojure.

10.1.4 Learning to love the brackets

The culture of programmers has always had a large element of whimsy and humor. One of the oldest jokes is that Lisp is an acronym for *Lots of Irritating Silly Parentheses* (instead of the more prosaic truth—that it's an abbreviation for List Processing). This rather self-deprecating joke is popular with some Lisp coders, partly because it points out the unfortunate truth that Lisp syntax has a reputation for being difficult to learn.

In reality, this hurdle is rather exaggerated. Lisp syntax is different from what most programmers are used to, but it isn't the obstacle that it's sometimes presented as. In addition, Clojure has several innovations that reduce the barrier to entry even further.

Let's take another look at the Hello World example. To call the function that returns the value "Hello World", we wrote this:

```
(hello)
```

If we want functions with arguments, then rather than having expressions such as `myFunction(someObj)`, in Clojure we write `(myFunction someObj)`. This syntax is called *Polish notation*, as it was developed by a Polish mathematicians in the early 20th century (it is also called *prefix notation*).

If you've studied compiler theory, you might wonder if there's a connection here to concepts like the abstract syntax tree (AST). The short answer is yes, there is. A Clojure (or other Lisp) program that is written in Polish notation (usually called an *s-expression* by Lisp programmers) can be shown to be a very simple and direct representation of the AST of that program.

NOTE

This relates back, once again, to the simple nature of the Clojure compiler. Compilation of Lisp code is a very cheap operation, because the structure is so close to the AST.

You can think of a Lisp program as being written in terms of its AST directly. There's no real distinction between a data structure representing a Lisp program and the code, so code and data are very interchangeable. This is the reason for the slightly strange notation — it's used by Lisp-like languages to blur the distinction between built-in primitives and user and library code.

This power is so great that it far outweighs the slight oddity of the syntax to the eyes of a newly arrived Java programmer.

Let's dive into some more of the syntax and start using Clojure to build real programs.

10.2 Looking for Clojure: syntax and semantics

In the previous section, you met the `(def)` and `(fn)` special forms (we also met `(defn)` but it's a macro, not a special form). There are a small number of other special forms that you need to know immediately to provide a basic vocabulary for the language. In addition, there are a large number of useful forms and macros, of which a greater awareness will develop with practice.

Clojure is blessed with a very large number of useful functions for doing a wide range of conceivable tasks. Don't be daunted by this — embrace it. Be happy that for many practical programming tasks you may face in Clojure, somebody else has already done the heavy lifting for you.

In this section, we'll cover the basic working set of special forms, then progress to Clojure's native data types (the equivalent of Java's collections). After that, we'll progress to a natural style for writing Clojure — one in which functions rather than variables have center stage. The object-oriented nature of the JVM will still be present beneath the surface, but Clojure's emphasis on functions has a power that is not as obviously present in purely OO languages - and which goes far beyond the basics of `map()`, `filter()` and `reduce()`.

10.2.1 Special forms bootcamp

Table 10.1 covers the definitions of some of Clojure's most commonly used special forms. To get best use of the table, skim through it now and refer back to it as necessary when you reach some of the examples in sections 10.3 onwards.

The table uses the traditional regular expression syntax notation where `?` represents a single optional value, and `*` represents zero or more values.

Table 10.1 Some of Clojure's basic special forms

Special form	Meaning
(def <symbol> <value?>)	Binds a symbol to a value (if provided). Creates a var corresponding to the symbol if necessary.
(fn <name?> [<arg>*] <expr>*)	Returns a function value that takes the specified args, and applies them to the exprs. Often combined with (def) into forms like (defn).
(if <test> <then> <else?>?)	If test evaluates to logical-true, evaluate and yield then. Otherwise, evaluate and yield else, if present.
(do <expr>*)	Evaluates the exprs in left-to-right order and yields the value of the last.
(let [<binding>*] <expr>*)	Aliases values to a local name and implicitly defines a scope. Makes the alias available inside all exprs within the scope of let.
(quote <form>)	Returns form as-is without evaluating anything. It takes a single form and ignores all other arguments.
(var <symbol>)	Returns the var corresponding to symbol (returns a Clojure JVM object, not a value).

This isn't an exhaustive list of special forms, and a high percentage of them have multiple ways of being used. Table 10.1 is a starter collection of basic use cases, and not anything comprehensive.

A couple of points deserve further explanation, as the structure of Clojure code can seem very different to Java code at first glance. First, the (do) form is one of the simplest ways to construct what would be a block of statements in Java.

Secondly, we need to dig a bit deeper into the distinction between a var, a value and the symbol that a value is (temporarily) bound to.

This simple code creates a Clojure var called `hi` - this is a JVM object (an instance of the type `clojure.lang.Var`) that lives in the heap, as all objects do - and binds it to a `java.lang.String` object containing "hello":

```
user=> (def hi "Hello")
#'user/hi
```

The var has a *symbol* `hi` - and it also has a *namespace* `user` that Clojure uses to organise programs - a bit like a Java package. If we use the symbol unadorned in the REPL then it evaluates to the value it is currently bound to:

```
user=> hi
"Hello"
```

In the (def) form, we bind a new symbol to a value, so in this code:

```
user=> (def bye hi)
#'user/bye
```

This means that the symbol `bye` is bound to the *value* currently bound to `hi`:

```
user=> bye
"Hello"
```

Effectively, in this simple form, `hi` is evaluated and the symbol is replaced with the value that results.

However, Clojure offers us more possibilities than just this. For example, the value that a symbol is bound to is just any JVM value. So, we can bind a symbol to the var we have created - because the var is itself a JVM object.

This is achieved using the `(var)` special form as shown here:

```
user=> (def bye (var hi))
#'user/bye

user=> bye
#'user/hi
```

This effectively uses the fact that Java / JVM objects are always handled by reference, as we can see in Figure 10.4.

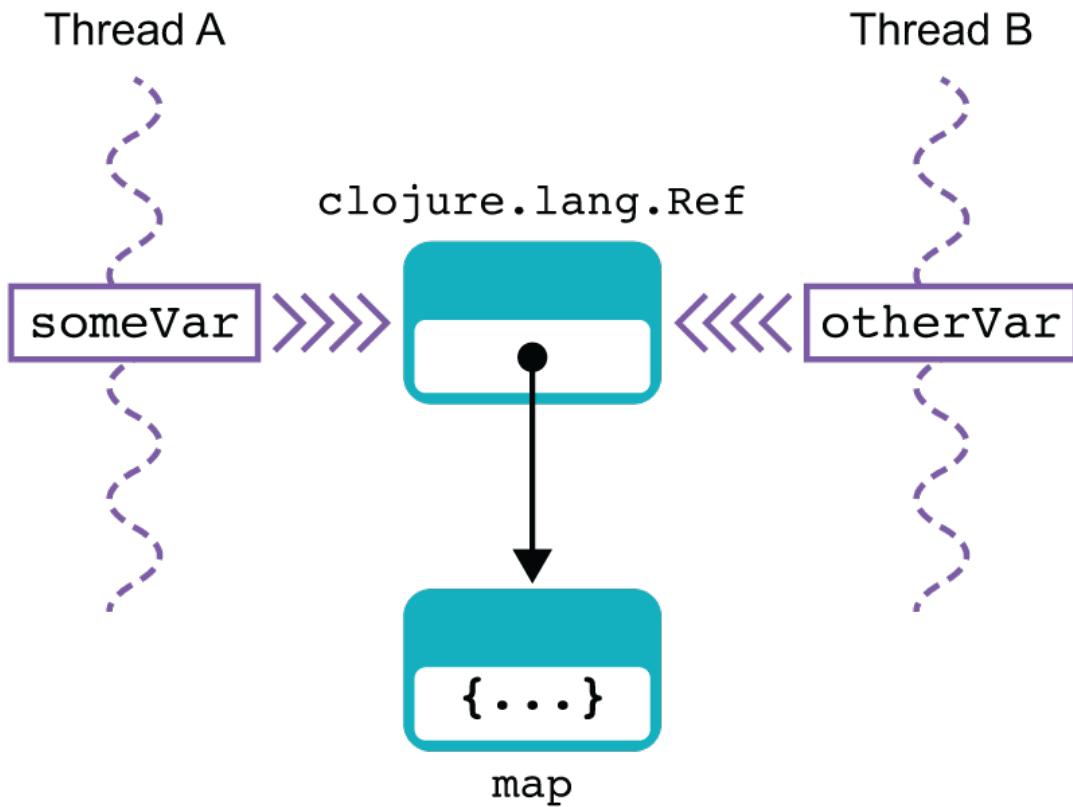


Figure 10.4 Clojure var acting by reference

To get back the value contained in a var, we can use the `(deref)` form (short for "dereference"), like this:

```
user=> (deref bye)
"Hello"
```

There is also a `(ref)` form that is used for safe concurrent programming in Clojure - we will meet it in Chapter 15.

From this distinction between a var and the value it is currently bound to, the `(quote)` form should be easier to understand - instead of evaluating the form it is passed, it simply returns a form comprising the unevaluated symbols.

Now that you have an appreciation of the syntax for some basic special forms, let's turn to Clojure's data structures and start to see how the forms can operate on data.

10.2.2 Lists, vectors, maps, and sets

Clojure has several native data structures. The most familiar is the *list*, which in Clojure is a singly linked list.

NOTE

In some respects, a Clojure list is similar to a `LinkedList` in Java, except that `LinkedList` is a doubly-linked list - where each element has a reference to both the next element and the previous.

Lists are typically surrounded with parentheses, which seemingly presents a slight syntactic hurdle because round brackets are also used for general forms. In particular, parentheses are used for evaluation of function calls. This leads to the following common beginner's syntax error:

```
user=> (1 2 3)
Execution error (ClassCastException) at user/eval1 (REPL:1).
class java.lang.Long cannot be cast to class clojure.lang.IFn
(java.lang.Long is in module java.base of loader 'bootstrap';
clojure.lang.IFn is in unnamed module of loader 'app')
```

The problem here is that, because Clojure is very flexible about its values, it's expecting a function value (or a symbol that resolves to one) as the first argument, so it can call that function and pass 2 and 3 as arguments; 1 isn't a value that is a function, so Clojure can't compile this form. We say that this *s-expression* is invalid, and recall that only valid *s-expressions* are Clojure forms.

The solution is to use the `(quote)` form that we met in the last section. This has a handy short form, which is `'`. This gives us these two equivalent ways of writing this list, which consists of the immutable list of 3 elements that are the numbers 1, 2 and 3.

```
1:22 user=> '(1 2 3)
(1 2 3)

1:23 user=> (quote (1 2 3))
(1 2 3)
```

Note that `(quote)` handles its arguments in a special way. In particular, there is no attempt made to evaluate the argument, so there's no error arising from a lack of a function value in the first slot.

Clojure has vectors, which are like arrays (in fact, it's not too far from the truth to think of lists as being basically like Java's `LinkedList` and vectors as like `ArrayList`). They have a convenient literal form that makes use of square brackets, so all of these are equivalent:

```
1:4 user=> (vector 1 2 3)
[1 2 3]

1:5 user=> (vec '(1 2 3))
[1 2 3]

1:6 user=> [1 2 3]
[1 2 3]
```

We've already met vectors. When we declared the Hello World function and others, we used a

vector to indicate the parameters that the declared function takes. Note that the form (`vec`) accepts a list and creates a vector from it, whereas (`vector`) is a form that accepts multiple individual symbols and returns a vector of them.

The function (`nth`) for collections takes two parameters: a collection and an index. It can be thought of as similar to the `get()` method from Java's `List` interface. It can be used on vectors and lists, but also on Java collections and even strings, which are treated as collections of characters. Here's an example:

```
1:7 user=> (nth '(1 2 3) 1)
2
```

Clojure also supports maps (which you can think of as being very similar to Java's `HashMap` - and they do in fact implement the `Map` interface), with this simple literal syntax:

```
{key1 value1 key2 value2}
```

To get a value back out of a map, the syntax is very simple:

```
user=> (def foo {"aaa" "111" "bbb" "2222"})
#'user/foo

user=> foo
{"aaa" "111", "bbb" "2222"}

user=> (foo "aaa")
①
"111"
```

- ① This syntax is equivalent to the use of a `get()` method in Java

As well as the `Map` interface, Clojure maps also implement the `IFn` interface, which is why they can be used in a form like (`foo "aaa"`) without a runtime exception.

One very useful stylistic point is the use of keys that have a colon in front of them. Clojure refers to these as *keywords*.

NOTE

The Clojure usage of "keyword" is of course very different from the meaning of that term in other languages (including Java) where the term means the parts of the language grammar that are reserved and not able to be used as identifiers.

Here are some useful points about keywords and maps to keep in mind:

- A keyword in Clojure is a function that takes one argument, which must be a map.
- Calling a keyword function on a map returns the value that corresponds to the keyword function in the map.
- When using keywords, there's a useful symmetry in the syntax, as (`my-map :key`) and (`:key my-map`) are both legal.

- As a value, a keyword returns itself.
- Keywords don't need to be declared or def'd before use.
- Remember that Clojure functions are values, and therefore are eligible to be used as keys in maps.
- Commas can be used (but aren't necessary) to separate key/value pairs, as Clojure considers them whitespace.
- Symbols other than keywords can be used as keys in Clojure maps, but the keyword syntax is extremely useful and is worth emphasizing as a style in your own code.

Let's see some of these points in action:

```
user=> (def martijn {:name "Martijn Verburg", :city "London",
:area "Finsbury Park"})
#'user/martijn

user=> (:name martijn) ①
"Martijn Verburg"

user=> (martijn :area) ②
"Finsbury Park"

user=> :area
:area

user=> :foo
:foo ③
```

- ➊ Calling the keyword function on the map
- ➋ Looking up the value associated to the keyword in the map
- ➌ Showing that when evaluated as a value, a keyword returns itself

In addition to map literals, Clojure also has a (`map`) function. But don't be caught out. Unlike (`list`), the (`map`) function doesn't produce a map. Instead, (`map`) applies a supplied function to each element in a collection in turn, and builds a new collection (actually a Clojure sequence, which you'll meet in detail in section 10.4) from the new values returned.

This is, of course, the Clojure equivalent to the `map()` method that you have already met from Java's Streams API.

```
user=> (def ben {:name "Ben Evans", :city "Barcelona", :area
"El Born"})
#'user/ben

user=> (def authors [ben martijn]) ①
#'user/authors

user=> (defn get-name [y] (:name y))
#'user/get-name

user=> (map get-name authors) ②
("Ben Evans" "Martijn Verburg")

user=> (map (fn [y] (:name y)) authors)
("Ben Evans" "Martijn Verburg") ③
```

- ① Create a vector of maps of author data
- ② Map the `get-name` function over the data
- ③ Alternate form using an inline function literal

There are additional forms of (`map`) that are able to handle multiple collections at once, but the form that takes a single collection as input is the most common.

Clojure also supports sets, which are very similar to Java's `HashSet`. They have a short form for data structure literals - which do not support repeated keys (unlike `HashSet`):

```
user=> #{ "a" "b" "c" }
#{ "a" "b" "c" }

user=> #{ "a" "b" "a" }
Syntax error reading source at (REPL:15:15).
Duplicate key: a
```

These data structures provide the fundamentals for building up Clojure programs.

One thing that may surprise the Java native is the lack of any immediate mention of objects as first-class citizens. This isn't to say that Clojure isn't object-oriented, but it doesn't see OO in quite the same way as Java. Java chooses to see the world in terms of statically typed bundles of data and code in explicit class definitions of user-defined data types. Clojure emphasizes the functions and forms instead, although these are implemented as objects on the JVM behind the scenes.

This philosophical distinction between Clojure and Java manifests itself in how code is written in the two languages, and to fully understand the Clojure viewpoint, it's necessary to write programs in Clojure and understand some of the advantages that deemphasizing Java's OO constructs brings.

10.2.3 Arithmetic, equality, and other operations

Clojure has no operators in the sense that you might expect them in Java. So how would you, for example, add two numbers? In Java it's easy:

```
3 + 4
```

But Clojure has no operators. We'll have to use a function instead:

```
(add 3 4)
```

①

- ① This code won't work as it stands, unless we supply an `add` function

That's all well and good, but we can do better. As there aren't any operators in Clojure, we don't

need to reserve any of the keyboard's characters to represent them. That means our function names can be more outlandish than in Java, so we can write this:

```
(+ 3 4)
```

①

- ① This is literally Polish notation as discussed earlier

Clojure's functions are in many cases *variadic* (they take a variable number of inputs), so you can, for example, write this:

```
(+ 1 2 3)
```

This will give the value 6.

For the equality forms (the equivalent of `equals()` and `==` in Java), the situation is a little more complex. Clojure has two main forms that relate to equality: `(=)` and `(identical?)`. Note that these are both examples of how the lack of operators in Clojure means that more characters can be used in function names. Also, `(=)` is a single equals sign, because there's not the same notion of assignment as in Java-like languages.

This bit of REPL code sets up a list, `list-int`, and a vector, `vect-int`, and applies equality logic to them:

```
1:1 user=> (def list-int '(1 2 3 4))
#'user/list-int

1:2 user=> (def vect-int (vec list-int))
#'user/vect-int

1:3 user=> (= vect-int list-int)
true

1:4 user=> (identical? vect-int list-int)
false
```

The key point is that the `(=)` form on collections checks to see whether the collections comprise the same objects in the same order (which is true for `list-int` and `vect-int`), whereas `(identical?)` checks to see if they're really the same object.

You might also notice that our symbol names don't use camel-case. This is usual for Clojure. Symbols are usually all in lowercase, with hyphens between words (sometimes called *kebab case*).

TRUE AND FALSE IN CLOJURE

Clojure provides two values for logical false: `false` and `nil`. Anything else is logical true (including the literal `true`). This parallels the situation in many dynamic languages (e.g. Javascript), but it's a bit strange for Java programmers encountering it for the first time.

With basic data structures and operators under our belts, let's put together some of the special forms and functions we've seen and write slightly longer example Clojure functions.

10.2.4 Working with functions in Clojure

In this section, we'll start dealing with some of the meat of Clojure programming. We'll start writing functions to act on data and bring Clojure's focus on functions to the fore. Next up are Clojure's looping constructs, then reader macros and dispatch forms. We'll round out the section by discussing Clojure's approach to functional programming, and its take on closures.

The best way to start doing all of this is by example, so let's get going with a few simple examples and build up toward some of the powerful functional programming techniques that Clojure provides.

SOME SIMPLE CLOJURE FUNCTIONS

Listing 10.1 defines three functions. Two of which are very simple functions of one argument; the third is a little more complex.

Listing 10.1 Defining simple Clojure functions

```
(defn const-fun1 [y] 1)

(defn ident-fun [y] y)

(defn list-maker-fun [x f]
  (map (fn [z] (let [w z]
                 (list w (f w)))
        )) x))
```

①
②
③

- ① The list-maker takes two arguments, the second of which is a function
- ② An inline, anonymous function
- ③ Make a list of two elements - the value, and the result of applying f to the value

In this listing, (`const-fun1`) takes in a value and returns 1, and (`ident-fun`) takes in a value and returns the very same value. Mathematicians would call these a *constant function* and the *identity function*. You can also see that the definition of a function uses vector literals to denote the arguments to a function, and for the (`let`) form.

The third function is more complex. The function (`list-maker-fun`) takes two arguments: first a vector of values to operate on, which is called `x`, and secondly a function (called `f`).

If we were to write it in Java, it might look a bit like this:

```
public List<Object> listMakerFun(List<Object> x,
                                    Function<Object, Object> f) {
    return x.stream()
        .map(o -> List.of(o, f.apply(o)))
        .collect(toList());
}
```

The role of the inline anonymous function in Clojure is played by the lambda expression in the Java code. However, it is important not to overstate the equivalence of these two code listings - Clojure and Java are *very* different languages.

NOTE Functions that take other functions as arguments are called **higher-order functions**. We'll meet them properly in Chapter 14.

Let's take a look at how (`list-maker-fun`) works:

Listing 10.2 Working with functions

```
user=> (list-maker-fun ["a"] const-fun1)
(("a" 1))

user=> (list-maker-fun ["a" "b"] const-fun1)
(("a" 1) ("b" 1))

user=> (list-maker-fun [2 1 3] ident-fun)
(2 2) (1 1) (3 3))

user=> (list-maker-fun [2 1 3] "a")
java.lang.ClassCastException: java.lang.String cannot be cast to
clojure.lang.IFn
```

Note that when you're typing these expressions into the REPL, you're interacting with the Clojure compiler. The expression `(list-maker-fun [2 1 3] "a")` fails to run (although it does compile) because `(list-maker-fun)` expects its second argument to be a function, which a string isn't. So although the Clojure compiler outputs bytecode for the form, it fails with a runtime exception.

NOTE In Java we can write valid code like `Integer.parseInt("foo")` which will compile fine, but will always fail at runtime. The Clojure situation is similar.

This example shows that when interacting with the REPL, you still have a certain amount of static typing in play. This is because Clojure isn't an interpreted language. Even in the REPL, every Clojure form that is typed is compiled to JVM bytecode and linked into the running system. The Clojure function is compiled to JVM bytecode when it's defined, so the `ClassCastException` occurs because of a static typing violation in the VM.

Listing 10.3 shows a longer piece of Clojure code, the Schwartzian transform. This is a piece of programming history, made popular by the Perl programming language in the 1990s. The idea

is to do a sort operation on a vector, based not on the provided vector, but on some property of the elements of the vector. The property values to sort on are found by calling a *keying function* on the elements.

The definition of the Schwartzian transform in listing 10.3 calls the keying function `key-fn`. When you actually want to call the `(schwartz)` function, you need to supply a function to use for keying. In listing 10.3, we use our old friend, `(ident-fun)`, from listing 10.1.

Listing 10.3 Schwartzian transform

```

user=> (defn schwartz [x key-fn]
  (map (fn [y] (nth y 0))
    (sort-by (fn [t] (nth t 1))
      (map (fn [z] (let [w z]
        (list w (key-fn w)))
      )) x))))
#'user/schwartz

user=> (schwartz [2 3 1 5 4] ident-fun)
(1 2 3 4 5)

user=> (apply schwartz [[2 3 1 5 4] ident-fun])
(1 2 3 4 5)

```

- ① Make a list consisting of pairs using the keying function
- ② Sort the pairs based on the values of the keying function
- ③ Construct a new list by reducing - taking only the original value from each pair

This code is performing three separate steps, which may seem a little inside-out at first glance. The steps are shown in Figure 10.5.

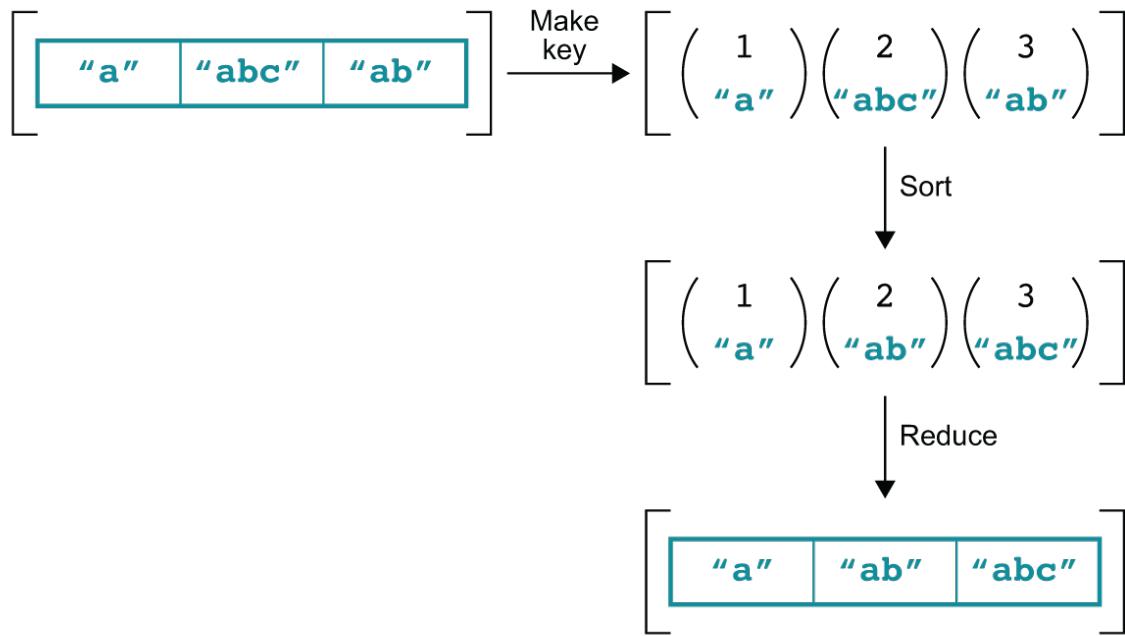


Figure 10.5 The Schwartzian transform

Note that in listing 10.3 we introduced a new form: `(sort-by)`. This is a function that takes two arguments: a function to use to do the sorting, and a vector to be sorted. We've also showcased the `(apply)` form, which takes two arguments: a function to call, and a vector of arguments to pass to it.

One amusing aspect of the Schwartzian transform is that the person for whom it was named was deliberately aping Lisp when he came up with the Perl version. Representing it in the Clojure code here means we've come full circle—back to a Lisp again!

The Schwartzian transform is a useful example that we'll refer back to later on. This is because it contains just enough complexity to demonstrate quite a few useful concepts.

Now, let's move on to discuss loops in Clojure, which work a bit differently than you may be used to.

10.2.5 Loops in Clojure

Loops in Java are a fairly straightforward proposition—the developer can choose from a `for`, a `while`, and a couple of other loop types. Usually central is the concept of repeating a group of statements until a condition (often expressed in terms of a mutable variable) is met.

This presents us with a slight conundrum in Clojure: how can we express a `for` loop when there are no mutable variables to act as the loop index? In more traditional Lisps, this is often solved by rewriting iterative loops into a form that uses recursion.

However, the JVM doesn't guarantee to optimize tail recursion (as is required by Scheme and other Lisps) so naively using recursion can cause the stack to blow up. We will have more to say about this issue in Chapter 14.

Instead, Clojure provides some useful constructions to allow looping without increasing the size of the stack. One of the most common is `loop-recur`. The next snippet shows how `loop-recur` can be used to build up a simple construction similar to a Java `for` loop.

```
(defn like-for [counter]
  (loop [ctr counter]                                ①
    (println ctr)
    (if (< ctr 10)
        (recur (inc ctr))                           ②
        ctr
      )))
```

- ① The loop entry point
- ② The recur point where we jump backwards

The `(loop)` form takes a vector of arguments of local names for symbols—effectively aliases as `(let)` does. Then, when execution reaches the `(recur)` form (which it will only do in this example if the `ctr` alias is less than 10), the `(recur)` causes control to branch back to the `(loop)` form, but with the new value specified.

This is similar to a rather primitive form of Java loop construction:

```
public int likeFor(int ctr) {
    LOOP: while (true) {
        System.out.println(ctr);
        if (ctr < 10) {
            ctr = ctr + 1;
            continue LOOP;
        } else {
            return ctr;
        }
    }
}
```

However, for a functional programmer the only very common reason to return early is if some condition is met. However, functions return the result of the last form evaluated, and `(if)` basically already does this for us.

In our example, we put the `(recur)` in the body of the `if` and the counter value in the `else` position.

This allows us to build up iteration-style constructs (such as the equivalent of Java's `for` and `while` loops), but to still have a functional flavor to the implementation.

We'll now turn to our next topic, which is a look at useful shorthand in Clojure syntax, to help make your programs even shorter and less verbose.

10.2.6 Reader macros and dispatch

Clojure has syntax features that surprise many Java programmers. One of them is the lack of operators. This has the side effect of relaxing Java's restrictions on which characters can be used in function names. You've already met functions such as (`identical?`), which would be illegal in Java, but we haven't addressed the issue of exactly which characters are and aren't allowed in symbols.

Table 10.2 lists the characters that aren't allowed in Clojure symbols. These are all characters that are reserved by the Clojure parser for its own use. They're usually referred to as *reader macros* and they are effectively a special character sequence which, when seen by the reader (first part of the Clojure compiler), modifies the reader's behavior.

For example, the `:` reader macro is how Clojure implements single-line comments. When the reader sees `:` it immediately ignores all remaining characters on this line, then resets to take the next line of input.

NOTE Later on, we will meet Clojure's general (or regular) macros. It is important not to confuse a reader macro with a regular macro.

Reader macros only exist for syntactical concision and convenience, not to provide a full general purpose metaprogramming capability.

Table 10.2 Reader macros

Character	Name	Meaning
'	Quote	Expands to <code>(quote)</code> . Yields the unevaluated form.
:	Comment	Marks a comment to end of line. Like <code>//</code> in Java.
\	Character	Produces a literal character. For example <code>\n</code> for newline.
@	Deref	Expands to <code>(deref)</code> , which takes in a var object and returns the value in that object (the opposite action of the <code>(var)</code> form). Has additional meaning in a transactional memory context (see Chapter 15).
^	Metadata	Attaches a map of metadata to an object. See the Clojure documentation for details.
`	Syntax-quote	Form of quote often used in macro definitions. See the macros section for details.
#	Dispatch	Has several different subforms. See table 10.3

The dispatch reader macro has several different subforms, depending on what follows the # character. Table 10.3 shows the different possible forms.

Table 10.3 The subforms of the dispatch reader macro

Dispatch form	Meaning
#'	Expands to (var).
#{ }	Creates a set literal, as discussed in section 10.2.2.
#()	Creates an anonymous function literal. Useful for single uses where (fn) is too wordy.
#_	Skips the next form. Can be used to produce a multiline comment, via #_(... multi-line ...).
#"<pattern>"	Creates a regular expression literal (as a java.util.regex.Pattern object).

Here are a couple of additional points that follow from the dispatch forms. The var-quote, #' , form explains why the REPL behaves as it does after a (def) :

```
1:49 user=> (def someSymbol)
#'user/someSymbol
```

The (def) form returns the newly created var object named someSymbol , which lives in the current namespace (which is user in the REPL), so #'user/someSymbol is the full value of what's returned from (def) .

The anonymous function literal #() also has a major innovation to reduce verboseness. This is to omit the vector of arguments, and instead use a special syntax to allow the Clojure reader to infer how many arguments are required for the function literal.

The syntax is %N where N is the number of the argument to the function.

Let's return to an earlier example and see how to use it with anonymous functions. Recall the (list-maker-fun) that takes two arguments (a list and a function) and creates a new list by applying the function to each element in turn:

```
(defn list-maker-fun [x f]
  (map (fn [z] (let [w z]
    (list w (f w)))
  )) x))
```

Rather than going to all the bother of defining a separate symbol, we can call this function with an inline function:

```
user=> (list-maker-fun ["a" "b"] (fn [x] x))
(("a" "a") ("b" "b"))
```

But we can go one step further, and use the more compact #() syntax:

```
user=> (list-maker-fun ["a" "b"] #(do %1))
(("a" "a") ("b" "b"))
```

This example is a little unusual, because we're using the `(do)` form we met back in the table of basic special forms, but it works.

Now, let's simplify `(list-maker-fun)` itself using the `#()` form:

```
(defn list-maker-fun [x f]
  (map #(list %1 (f %1)) x))
```

The Schwartzian transform also makes an excellent use case to see how to use this syntax in a more complex example:

Listing 10.4 Rewritten Schwartzian transform

```
(defn schwartz [x key-fn]
  (map #(nth %1 0)
    (sort-by #(nth %1 1)
      (map #(let [w %1]
        (list w (key-fn w))
      ) x))))
```

①
①
①

- ① Anonymous function literals corresponding to the 3 steps

The use of `%1` as a placeholder for a function literal's argument (and `%2`, `%3`, and so on for subsequent arguments) makes the usage really stand out, and makes the code a lot easier to read. This visual clue can be a real help for the programmer, similar to the arrow symbol used in lambda expressions in Java.

As you've seen, Clojure relies heavily on the concept of functions as the basic unit of computation, rather than on objects, which are the staple of languages like Java. The natural setting for this approach is functional programming, which is our next topic.

10.3 Functional programming and closures

We're now going to turn to the scary world of functional programming in Clojure. Or rather, we're *not*, because it's just not that scary. In fact, we've been doing functional programming for this entire chapter; we just didn't tell you, in order to not put you off.

As we mentioned in section 8.XXX, functional programming is a somewhat nebulous concept - all it can be relied upon to mean is that a function is a value. A function can be passed around, placed in variables and manipulated, just like `2` or `"Hello world"`. But so what? We did that back in our very first example: `(def hello (fn [] "Hello world"))`. We created a function (one that takes no arguments and returns the string `"Hello world"`) and bound it to the symbol `hello`. The function was just a value, not fundamentally different for a value like `2`.

In section 10.3.1, we introduced the Schwartzian transform as an example of a function that takes another function as an input value. Again, this is just a function taking a particular type as one of its input arguments. The only thing that's slightly different about is that the type it's taking is a function.

It's probably also a good time to introduce the `(filter)` form, which should remind you of the similarly-named method in Java Streams:

```
user=> (defn gt4 [x] (> x 4))
#'user/gt4
user=> (filter gt4 [1 2 3 4 5 6])
(5 6)
```

There is also the `(reduce)` form, to complete the set of filter-map-reduce operations. It is most commonly seen in two variants, one that takes an initial starting value (sometimes called a "zero") and one that doesn't:

```
user=> (reduce + 1 [2 3 4 5])
15
user=> (reduce + [1 2 3 4 5])
15
```

What about closures? Surely they're really scary, right? Well, not so much. Let's take a look at a simple example that should hopefully remind you of some of the examples we did for Kotlin in chapter 9:

```
1:5 user=> (defn adder [constToAdd] #(+ constToAdd %1))
#'user/adder

1:6 user=> (def plus2 (adder 2))
#'user/plus2

1:7 user=> (plus2 3)
5

1:8 user=> 1:9 user=> (plus2 5)
7
```

You first set up a function called `(adder)`. This is a function that makes other functions. If you're familiar with the Factory Method pattern in Java, you can think of this as kind-of a Clojure equivalent. There's nothing strange about functions that have other functions as their return values—this is a key part of the concept that functions are just ordinary values.

Notice that this example uses the shorthand form `#()` for an anonymous function literal. The function `(adder)` takes in a number and returns a function, and the function returned from `(adder)` takes one argument.

You then use `(adder)` to define a new form: `(plus2)`. This is a function that takes one numeric argument and adds 2 to it. That means the value that was bound to `constToAdd` inside `(adder)` was 2. Now let's make a new function:

```

1:13 user=> (def plus3 (adder 3))
 #'user/plus3

1:14 user=> (plus3 4)
 7

1:15 user=> (plus2 4)
 6

```

This shows that you can make a different function, `(plus3)`, that has a different value bound to `constToAdd`. We say that the functions `(plus3)` and `(plus2)` have *captured*, or *closed over* a value from their environment. Note that the values that were captured by `(plus3)` and `(plus2)` were different, and that defining `(plus3)` had no effect on the value captured by `(plus2)`.

Functions that close over some values in their environment are called *closures*; `(plus2)` and `(plus3)` are examples of closures. The pattern whereby a function-making function returns another, simpler function that has closed over something is a very common one in languages that have closures.

NOTE

Remember that although Clojure will compile any syntactically form, the program will throw a runtime exception if a function is called with the wrong number of arguments. A 2-argument function could not be used in a place where a 1-argument function was expected.

We will have a lot more to say about functional programming in context in Chapter 13.

Now let's turn to a powerful Clojure feature — sequences.

10.4 Introducing Clojure sequences

Clojure has a powerful core abstraction the called the *sequence*, or more usually *seq*.

NOTE

Sequences are a major part of writing Clojure code that utilizes the strengths of the language, and they'll provide an interesting contrast to how Java handles similar concepts.

The seq type roughly corresponds to collections and / or iterators in Java, but seqs have somewhat different properties.

The fundamental idea is that seqs essentially merge some of the features of both Java types into one concept. This is motivated by wanting three things:

- Immutability, allowing the seqs to be passed around between functions (and threads) without problems
- A more robust iterator-like abstraction, especially for multipass algorithms
- The possibility of *lazy sequences* (more on these later)

Of these three things, the one that Java programmers sometimes struggle with the most is the immutability. The Java concept of an iterator is inherently mutable, partly because it does not provide a cleanly separable interface.

In fact, Java's `Iterator` violates the Single Responsibility Principle - because `next()` does *two* things logically distinct things when called:

- It returns the currently pointed-at element
- It mutates the iterator, by advancing the element pointer

The seq is based around functional ideas, and avoids the mutation by dividing up the capabilities of `hasNext()` and `next()` in a different way. Let's meet a slightly simplified version of another of Clojure's most important interfaces, `clojure.lang.ISeq`:

```
interface ISeq {
    Object first();
    ①
    ISeq rest();
    ②
}
```

- ① Return the object that is first in the seq
- ② Return a new seq that contains all the elements of the old seq, *except* the first

Now, the seq is never mutated - instead a new seq value is created every time we call `rest()` - which is when would have stepped the iterator to the next value. Let's look at some code to show how we might implement this in Java:

```

public class ArraySeq implements ISeq {
    private final int index;                                ①
    private final Object[] values;                           ①

    private ArraySeq(int index, Object[] values) {
        this.index = index;
        this.values = values;
    }

    public static ArraySeq of(List<Object> objects) {      ②
        if (objects == null || objects.size() == 0) {
            return Empty.of();
        }
        return new ArraySeq(0, objects.toArray());
    }

    @Override
    public Object first() {
        return values[index];
    }

    @Override
    public ISeq rest() {
        if (index >= values.length - 1) {                      ③
            return Empty.of();
        }
        return new ArraySeq(index + 1, values);
    }

    public int count() {
        return values.length - index;
    }
}

```

- ① Final fields
- ② Factory method that takes a List
- ③ Need an empty implementation as well

As you can see, we need a special-case seq for the end of the sequence. Let's represent it as an inner class within `ArraySeq`:

```

public static class Empty extends ArraySeq {
    private static Empty EMPTY = new Empty(-1, null);

    private Empty(int index, Object[] values) {
        super(index, values);
    }

    public static Empty of() {
        return EMPTY;
    }

    @Override
    public Object first() {
        return null;
    }

    @Override
    public ISeq rest() {
        return of();
    }

    public int count() {
        return 0;
    }
}

```

Let's see this in action:

```

ISeq seq = ArraySeq.of(List.of(10000,20000,30000));
var o1 = seq.first();
var o2 = seq.first();
System.out.println(o1 == o2);

```

As expected, calls to `first()` are *idempotent* - they do not change the seq and will repeatedly return the same value.

Let's look at how we'd write a loop in Java using `ISeq`:

```

while (seq.first() != null) {
    System.out.println(seq.first());
    seq = seq.rest();
}

```

This example shows how we deal with one objection that some Java programmers sometimes have with the immutable seq approach - "what about all the garbage?"

It's true that each call to `rest()` will create a new seq - which is an object. However, if you look closely at the implementing code you can see that we're careful not to duplicate `values` - the array storage. Copying that would be expensive, so we don't do that.

All we're really creating at each step is a tiny object that contains an int and a reference to an object. If these temporaries aren't stored anywhere then they'll fall out of scope as we walk down the seq, and very quickly become eligible for garbage collection.

NOTE The method bodies for `Empty` do not refer to either `index` or `values` - so we are free to use special values (-1 and null), which would not be able to be reached by any other instance of `ArrayList` - this is a debugging aid.

Let's switch back into Clojure now that we've explained some of the theory of seqs using Java.

NOTE The real `ISeq` interface that all Clojure sequences implement is a little more complex than the version we've met so far, but the basic intent is the same.

Some core functions that relate to sequences are shown in table 10.4. Note that none of these functions will mutate their input arguments; if they need to return a different value, it will be a different seq.

Table 10.4 Basic sequence functions

Function	Effect
<code>(seq <coll>)</code>	Returns a seq that acts as a "view" onto the collection acted upon.
<code>(first <coll>)</code>	Returns the first element of the collection, calling <code>(seq)</code> on it first if necessary. Returns nil if the collection is nil.
<code>(rest <coll>)</code>	Returns a new seq, made from the collection, minus the first element.
<code>(seq? <o>)</code>	Returns true if o is a seq (meaning, if it implements <code>ISeq</code>).
<code>(cons <elt> <coll>)</code>	Returns a seq made from the collection, with the additional element prepended.
<code>(conj <coll> <elt>)</code>	Returns a new collection with the new element added to the appropriate end—the end for vectors and the head for lists.
<code>(every? <pred-fn> <coll>)</code>	Returns true if (<code>pred-fn</code>) returns logical-true for every item in the collection.

Here are a few examples:

```

1:1 user=> (rest '(1 2 3))
(2 3)

1:2 user=> (first '(1 2 3))
1

1:3 user=> (rest [1 2 3])
(2 3)

1:13 user=> (seq ())
nil

1:14 user=> (seq [])
nil

1:15 user=> (cons 1 [2 3])
(1 2 3)

1:16 user=> (every? is-prime [2 3 5 7 11])
true

```

One important point to note is that Clojure lists are their own seqs, but vectors aren't. In theory, that would mean that you shouldn't be able to call `(rest)` on a vector. The reason you're able to is that `(rest)` acts by calling `(seq)` on the vector before operating on it.

NOTE

Many of the sequence functions take more general objects than seqs, and will call `(seq)` on them before they begin.

In the next section, we're going to explore some of the basic properties and uses of the seq abstraction, paying special attention to variadic functions. Later on, in Chapter 14, we'll meet *lazy sequences* - a very important functional technique.

10.4.1 Sequences and variable-arity functions

There is one powerful feature of Clojure's approach to functions that we've delayed discussing fully until now. This is the natural ability to easily have variable numbers of arguments to functions, sometimes called the *arity* of functions. Functions that accept variable numbers of parameters are called *variadic*, and they are frequently used when operating on seqs.

NOTE

Java supports variadic methods, with a syntax in which the final parameter of a method is shown with ... on the type, to indicate that any number of parameters of that type are allowed at the end of the parameter list.

As a trivial example, consider the constant function `(const-fun1)` that we discussed in listing 9.1. This function takes in a single argument and discards it, always returning the value 1. But consider what happens when you pass more than one argument to `(const-fun1)`:

```

user=> (const-fun1 2 3)
java.lang.IllegalArgumentException:
    Wrong number of args (2) passed to: user$const-fun1 (repl-1:32)

```

The Clojure compiler cannot enforce compile time static checks on the number (and types) of arguments passed to `(const-fun1)`, and instead we have to risk runtime exceptions.

This seems overly restrictive - especially for a function that simply discards all of its arguments and returns a constant value. What would a function that could take any number of arguments look like in Clojure?

Listing 10.6 shows how to do this for a version of the `(const-fun1)` constant function from earlier in the chapter. We've called it `(const-fun-arity1)`, for *constant function 1* with variable *arity*.

NOTE This is in fact a homebrew version of the `(constantly)` function provided in the Clojure standard function library.

Listing 10.5 Variable arity function

```
user=> (defn const-fun-arity1
  ([] 1)
  ([x] 1)
  ([x & more] 1)
)
#'user/const-fun-arity1

user=> (const-fun-arity1)
1

user=> (const-fun-arity1 2)
1

user=> (const-fun-arity1 2 3 4)
1
```

- ① Multiple function definitions with different signatures

The key is that the function definition is followed not by a vector of function parameters and then a form defining the behavior of the function. Instead, there is a list of pairs, with each pair consisting of a vector of parameters (effectively the signature of this version of the function) and the implementation for this version of the function.

This can be thought of as a similar concept to method overloading in Java. The usual convention is to define a few special-case forms (that take none, one, or two parameters) and an additional form that has as its last parameter a seq. In listing 10.6 this is the form that has the parameter vector of `[x & more]`. The & sign indicates that this is the variadic version of the function.

Sequences are a very powerful Clojure innovation. In fact, a large part of learning to think in Clojure is to start thinking about how the seq abstraction can be put to use to solve your specific coding problems.

Another important innovation in Clojure is the integration between Clojure and Java, which is the subject of the next section.

10.5 Interoperating between Clojure and Java

Clojure was designed from the ground up to be a JVM language and to not attempt to completely hide the JVM character from the programmer. These specific design choices are apparent in a number of places. For example, at the type-system level, Clojure's lists and vectors both implement `List` — the standard interface from the Java collections library. In addition, it's very easy to use Java libraries from Clojure and vice versa.

These properties are extremely useful, as it means that Clojure programmers can make use of the rich variety of Java libraries and tooling, as well as the performance and other features of the JVM.

In this section, we'll cover a number of aspects of this interoperability decision, specifically:

- Calling Java from Clojure
- How Java sees the type of Clojure functions
- Clojure proxies
- Exploratory programming with the REPL
- Calling Clojure from Java

Let's start exploring this integration by looking at how to access Java methods from Clojure.

10.5.1 Calling Java from Clojure

Consider this piece of Clojure code being evaluated in the REPL:

```
1:16 user=> (defn lenStr [y] (.length (.toString y)))
#'user/lenStr

1:17 user=> (schwartz ["bab" "aa" "dgfwg" "droopy"] lenStr)
("aa" "bab" "dgfwg" "droopy")
```

In this snippet, we've used the Schwartzian transform to sort a vector of strings by their lengths. To do that, we've used the forms `(.toString)` and `(.length)`, which are Java methods. They're being called on the Clojure objects. The period at the start of the symbol means that the runtime should invoke the named method on the next argument. This is achieved by the use of another macro that we haven't met yet - `(.)` - behind the scenes.

Recall that all Clojure values defined by `(def)` or a variant of it are placed into instances of `clojure.lang.Var`, which can house any `java.lang.Object`, so any method that can be called on `java.lang.Object` can be called on a Clojure value. Some of the other forms for interacting with the Java world are

```
(System/getProperty "java.vm.version")
```

for calling static methods (in this case the `System.getProperty()` method) and

```
Boolean/TRUE
```

for accessing static public variables (such as constants).

The familiar "Hello World" example looks like this:

```
user=> (.println System/out "Hello World")
Hello World
nil
```

and note that the final `nil` is because, of course, all Clojure forms must return a value - even if they are a call to a `void` Java method.

In these last three examples, we've implicitly used Clojure's namespaces concept. These are similar to Java packages, and have mappings from shorthand forms to Java package names for common cases, such as the preceding ones.

10.5.2 The nature of Clojure calls

A function call in Clojure is compiled to a JVM method call. The JVM does not guarantee to optimize away tail recursion, which Lisps (especially Scheme implementations) usually do. Some other Lisp dialects on the JVM take the viewpoint that they want true tail recursion and so are prepared to have a Lisp function call not be exactly equivalent to a JVM method call under all circumstances. Clojure, however, fully embraces the JVM as a platform, even at the expense of full compliance with usual Lisp practice.

If you want to create a new instance of a Java object and manipulate it in Clojure, you can easily do so by using the `(new)` form. This has an alternative short form, which is the class name followed by the full stop, which boils down to another use of the `(.)` macro:

```
(import '(java.util.concurrent CountDownLatch LinkedBlockingQueue))
(def cdl (new CountDownLatch 2))
(def lbq (LinkedBlockingQueue.))
```

Here we're also using the `(import)` form, which allows multiple Java classes from a single package to be imported in just one line.

We mentioned earlier that there's a certain amount of alignment between Clojure's type system and that of Java. Let's take a look at this concept in a bit more detail.

10.5.3 The Java type of Clojure values

From the REPL, it's very easy to take a look at the Java types of some Clojure values:

```
1:8 user=> (.getClass "foo")
java.lang.String

1:9 user=> (.getClass 2.3)
java.lang.Double

1:10 user=> (.getClass [1 2 3])
clojure.lang.PersistentVector

1:11 user=> (.getClass '(1 2 3))
clojure.lang.PersistentList

1:12 user=> (.getClass (fn [] "Hello world!"))
user$eval110$fn__111
```

The first thing to notice is that all Clojure values are objects; the primitive types of the JVM aren't exposed by default (although there are ways of getting at the primitive types for the performance-conscious). As you might expect, the string and numeric values map directly onto the corresponding Java reference types (`java.lang.String`, `java.lang.Double`, and so on).

The anonymous "Hello world!" function has a name that indicates that it's an instance of a dynamically generated class. This class will implement the interface `IFn`, which is the very important interface that Clojure uses to indicate that a value is a function.

As we discussed a bit earlier on, seqs implement the `ISeq` interface. They will typically be one of the concrete subclasses of the abstract `ASeq` or the lazy implementation, `LazySeq` (we'll meet laziness in Chapter 14 when we talk about advanced functional programming).

We've looked at the types of various values, but what about the storage for those values? As we mentioned at the start of this chapter, `(`def`)` binds a symbol to a value, and in doing so creates a var. These vars are objects of type `clojure.lang.Var` (which implements `IFn` amongst other interfaces).

10.5.4 Using Clojure proxies

Clojure has a powerful macro called `(proxy)` that enables you to create a bona fide Clojure object that extends a Java class (or implements an interface). For example, the next listing revisits an earlier example (listing 5.XX), but the heart of the execution example is now done in a fraction of the code, due to Clojure's more compact syntax.

Listing 10.6 Listing 9.7 Revisiting scheduled executors

```
(import '(java.util.concurrent Executors LinkedBlockingQueue TimeUnit))

(def stpe (Executors/newScheduledThreadPool 2))           ①

(def lbq (LinkedBlockingQueue.))

(def msgRdr (proxy [Runnable] []
  (run [] (.println System/out (.toString (.poll lbq))))))

(def rdrHndl
  (.scheduleAtFixedRate stpe msgRdr 10 10 TimeUnit/MILLISECONDS))
```

- ① Factory method to create an Executor
- ② Define an anonymous implementation of Runnable

The general form of (proxy) is

```
(proxy [<superclass/interfaces>] [<args>] <impls of named functions>+)
```

The first vector argument holds the interfaces that this proxy class should implement. If the proxy should also extend a Java class (and it can, of course, only extend one Java class), that class name must be the first element of the vector.

The second vector argument comprises the parameters to be passed to a superclass constructor. This is quite often the empty vector, and it will certainly be empty for all cases where the (proxy) form is just implementing Java interfaces.

After these two arguments come the forms that represent the implementations of individual methods, as required by the interfaces or superclasses specified.

In our example, the proxy only needs to implement `Runnable`, so that is the only symbol in the first vector of arguments. No superclass parameters are needed, so the second vector is empty (as it very often is).

Following the two vectors, comes a list of forms that define the methods that the proxy will implement. In our case, that is just `run()` - and we give it the definition `(run [] (.println System/out (.toString (.poll lbq))))`

This is, of course, just the Clojure way of writing this bit of Java:

```
public void run() {
    System.out.println(lbq.poll().toString());
}
```

The (proxy) form allows for the simple implementation of any Java interface. This leads to an intriguing possibility—that of using the Clojure REPL as an extended playpen for experimenting

with Java and JVM code.

10.5.5 Exploratory programming with the REPL

The key concept of exploratory programming is that with less code to write, due to Clojure's syntax, and the live, interactive environment that the REPL provides, the REPL can be a great environment for exploring not only Clojure programming, but for learning about Java libraries as well.

Let's consider the Java list implementations. They have an `iterator()` method that returns an object of type Iterator. But Iterator is an interface, so you might be curious about what the real implementing type is. Using the REPL, it's easy to find out:

```
1:41 user=> (import '(java.util ArrayList LinkedList))
java.util.LinkedList

1:42 user=> (.getClass (.iterator (ArrayList.)))
java.util.ArrayList$Itr

1:43 user=> (.getClass (.iterator (LinkedList.)))
java.util.LinkedList$ListItr
```

The `(import)` form brings in two different classes from the `java.util` package. Then you can use the `getClass()` Java method from within the REPL just as you did in section 10.5.2. As you can see, the iterators are actually provided by inner classes. This perhaps shouldn't be surprising; as we discussed in section 10.4, iterators are tightly bound up with the collections they come from, so they may need to see internal implementation details of those collections.

Notice that in the preceding example, we didn't use a single Clojure construct — just a little bit of syntax. Everything we were manipulating was a true Java construct. Let's suppose, though, that you wanted to use a different approach and use the powerful abstractions that Clojure brings within a Java program. The next subsection will show you just how to accomplish this.

10.5.6 Using Clojure from Java

Recall that Clojure's type system is closely aligned with Java's. The Clojure data structures are all true Java collections that implement the whole of the mandatory part of the Java interfaces. The optional parts aren't usually implemented, as they're often about mutation of the data structures, which Clojure doesn't support.

This alignment of type systems opens the possibility of using Clojure data structures in a Java program. This is made even more viable by the nature of Clojure itself — it's a compiled language with a calling mechanism that matches that of the JVM. This minimizes the runtime aspects and means a class obtained from Clojure can be treated almost like any other Java class. Interpreted languages would find it a lot harder to interoperate and would typically require a minimal non-Java language runtime for support.

The next example shows how Clojure's seq construct can be used on an ordinary Java string. For this code to run, `clojure.jar` will need to be on the classpath:

```
ISeq seq = StringSeq.create("foobar");

while (seq != null) {
    Object first = seq.first();
    System.out.println("Seq: " + seq + " ; first: " + first);
    seq = seq.next();
}
```

The preceding code snippet uses the factory method `create()` from the `StringSeq` class. This provides a seq view on the character sequence of the string. The `first()` and `next()` methods return new values, as opposed to mutating the existing seq, just as we discussed in section 10.4.

In the next section, we'll move on to talk Clojure's macros. This is a very powerful technique that allows the experienced programmer to effectively modify the Clojure language itself. This capability is common in languages like Lisp but rather alien to Java programmers, so it warrants an entire section to itself.

10.6 Macros

In Chapter 8, we discussed the rigidity the language grammar of Java. By contrast, Clojure provides and actively encourages *macros* as a mechanism to provide a much more flexible approach - allowing the programmer to write more-or-less-ordinary program code that behaves in the same way as built-in language syntax.

NOTE

Many languages have macros (including C++) and they mostly all operate in a roughly similar way - by providing a special phase of source code compilation - often the very first phase.

For example, in the C language, the first step is *preprocessing* which removes comments, inlines included files and expands macros - which are the different types of *preprocessor directives* such as `#include` and `#define`.

However, while C macros were very powerful, they also make it possible for engineers to produce some very subtly confusing code which is hard to understand and debug. To avoid this complexity, the Java language never implemented a macro system or a preprocessor.

C macros work by providing very simple text replacement capabilities during the preprocessing phase. Clojure macros are safer, because they work within the syntax of Clojure itself. Effectively, they allow the programmer to create a special kind of function that is evaluated (in a special way) at compile time. This means that the macro can transform source code during compilation - during what is referred to as *macro expansion time*.

NOTE The key to the power of macros is the fact that Clojure code is written down as a valid Clojure data structure - specifically as a list of forms.

We say that Clojure, like other Lisps (and a few other languages) is *homoiconic* - which means that programs are represented in the same way as data. Other programming languages, like Java, write their source code as a string and without parsing that string in a Java compiler the structure of the program cannot be determined.

Recall that Clojure compiles source code as it is encountered. Many Lisps are interpreted languages, but Clojure is not. Instead, when Clojure source code is loaded, it is compiled *on the fly* into JVM bytecode. This can give the superficial impression that Clojure is interpreted, but the (very simple) Clojure compiler is hiding just below the surface.

NOTE A Clojure form is a list - and a macro is essentially a function that does not evaluate its arguments but instead manipulates them to return another list which will then be compiled as a Clojure form.

To demonstrate this, let's try to write a macro form that acts like the opposite of `(if)` - in some languages this would be represented with the `unless` keyword, so in Clojure it will be an `(unless)` form.

So what we want is a form that looks like `(if)` but behaves as the logical opposite:

```
user=> (def test-me false)
#'user/test-me

user=> (unless test-me "yes")
"yes"

user=> (def test-me true)
#'user/test-me

user=> (unless test-me "yes")
nil
```

Note that we don't provide the equivalent of an `else` condition. This somewhat simplifies the example and "unless ... else" sounds weird anyway. In our examples, if the `unless` logical test fails, then the form evaluates to `nil`.

If we try to write this using `(defn)` then we can write a simple first attempt like this (spoiler: It won't actually work properly):

```

user=> (defn unless [p t]
  (if (not p) t))
#'user/unless

user=> (def test-me false)
#'user/test-me

user=> (unless test-me "yes")
"yes"

user=> (def test-me true)
#'user/test-me

user=> (unless test-me "yes")
nil

```

This seems fine. However, consider that we want (`unless`) to work the same way as (`if`) - in particular the `then` form should only evaluated if the boolean predicate condition is true. In other words, for (`if`) we see this behavior:

```

user=> (def test-me true)
#'user/test-me

user=> (if test-me (do (println "Test passed") true))
Test passed
true

user=> (def test-me false)
#'user/test-me

user=> (if test-me (do (println "Test passed") true))
nil

```

When we try to use our (`unless`) function in the same way then the problem becomes clear:

```

user=> (def test-me false)
#'user/test-me

user=> (unless test-me (do (println "Test passed") true))
Test passed
true

user=> (def test-me true)
#'user/test-me

user=> (unless test-me (do (println "Test passed") true))
Test passed
nil

```

Regardless of whether the predicate is true or false, the `then` form is still evaluated, and as it is (`println`) in our example, it produces output - which provides the clue that lets us know that the evaluation is taking place.

To solve this problem, we need to handle the forms that we are passed *without evaluating them*. This is essentially a (slightly different) kind of the laziness concept that is so important in functional programming (and which we will describe in detail in Chapter 14).

The special form (`defmacro`) is used to declare a new macro, like this:

```
(defmacro unless [p t]
  (list 'if (list 'not p) t))
```

Let's see if it does the right thing:

```
user=> (def test-me true)
#'user/test-me

user=> (unless test-me (do (println "Test passed") true))
nil

user=> (def test-me false)
#'user/test-me

user=> (unless test-me (do (println "Test passed") true))
Test passed
true
```

This now behaves as we want it to - essentially the `(unless)` form now looks and behaves just like the builtin `(if)` special form.

As you can see, one of the drawbacks of writing macros is that a lot of quoting is involved. The macro transforms its arguments to a new Clojure form at compile-time - so it is natural that the output should be a `(list)`.

The list contains Clojure symbols that will be evaluated at runtime, so anything that we do not explicitly need to evaluate during macro expansion must be quoted. This relies upon the fact that macros receive their arguments at compile time, so they are available as unevaluated data.

In our example, we need to quote everything that is **not** one of our arguments - these will be string-replaced as symbols during expansion. This gets pretty cumbersome fairly quickly. Can we do better?

Let's meet a helpful tool that might point us in the right direction. When writing or debugging macros the `(macroexpand-1)` form can be very useful. If this form is passed a macro form then it expands the macro and returns the expansion. If the passed form is not a macro, then it just returns the form. For example:

```
user=> (macroexpand-1 '(unless test-me (do (println "Test passed") true)))
(if (not test-me) (do (println "Test passed") true))
```

What we would really like is the ability to write macros that look like their macro-expanded form, and without the huge amount of quoting that we've seen in examples so far.

NOTE

Full macro expansion, using the form `(macroexpand)` is then constructed by just repeatedly calling the former, simpler form. When applying `(macroexpand-1)` is a no-op, then macro expansion is over.

The key to this capability is the special reader macro ` - which is pronounced "syntax-quote", and which we previewed earlier in the chapter as part of the section about reader macros.

The way that the syntax quoting reader macro ` works is that it basically quotes everything in the following form. If you want something to *not* be quoted then you have to use the syntax-unquote (~) operator to exempt a value from syntax quoting.

This means that our example macro (unless) can be written as:

```
(defmacro unless [p t]
  `(~(if (not ~p) ~t))
```

and this form is now much clearer, and closer to the form we see when macro expanding. The ~ character provides a nice visual clue to let us know that those symbols will be replaced when the macro is expanded. This fits nicely with the idea of a macro as a compile-time code template.

Along with syntax quote and unquote, there are some important special variables that are sometimes used in macro definitions. Of these, two of the most common are:

- &form - the expression that is being invoked
- &env - a map of local bindings at the point of macro expansion

Full details of the information that can be obtained from each special variable can be found in the Clojure documentation.

We should also note that care needs to be taken when writing Clojure macros. For example, it is possible to create macros that create recursive expansions that do not terminate and instead *diverge*. For example:

```
(defmacro diverge [t]
  `((diverge ~t) (diverge ~t)))
#'user/diverge

user=> (diverge true)
Syntax error (StackOverflowError) compiling at (REPL:1:1).
null
```

As a final example, let's confirm that macros do in fact operate at compile time, by constructing a macro that essentially acts as a closure that bridges from compile to runtime:

```

user=> (defmacro print-msg-with-compile []
  (let [num (System/currentTimeMillis)]
    `(fn [t#] (println t# " " ~num)))
#'user/print-msg-with-compile

user=> (def p1 (print-msg-with-compile))
#'user/p1

user=> (p1 "aaa")
aaa 1603437421852
nil

user=> (p1 "bbb")
bbb 1603437421852
nil

```

Notice how the `(let)` form is evaluated at compile time, so the value of `(System/currentTimeMillis)` is captured when the macro is evaluated, and bound to the symbol `num`, and then replaced in the expanded form with the value that was bound - effectively a constant determined at compile time.

Even though we have only introduced macros at the very end of this chapter, macros are actually all around us in Clojure. In fact, much of the Clojure standard library is implemented as macros. The well-grounded developer can learn a lot by spending some time reading the source of the standard library and observing how key parts of it have been written.

At this point, a word of warning is also timely. Macros are a very powerful technique, and there is a temptation (just as there is with other techniques that "level up" a programmer's thinking) that some developers can fall prey to. This is the tendency to overuse the technique including when it is not strictly necessary.

To guard against this, there are some simple general rules for the use of Clojure macros which we highly recommend that you follow:

- Never write a macro when the goal can be accomplished with a function
- Write a macro to implement a feature, capability or pattern that is not already present in the language or standard library

The first of these is, of course, merely the old adage that "just because you *can* do something doesn't mean that you *should*" in a different guise.

The second is a reminder that macros exist for a reason - there are things that you can do with them that cannot really be done in any other way. A proficient Clojure programmer will be able to use macros to great effect where appropriate.

Beyond macros, there is still more to learn about Clojure - such as the language's approach to dynamic runtime behavior. In Java this is usually handled using class and interface inheritance and virtual dispatch, but these are fundamentally object-oriented concepts and are not a particularly good fit for Clojure.

Instead, Clojure uses *protocols* and *datatypes* - along with the proxies that we have already met - to provide much of this flexibility. There are even more possibilities, such as custom dispatch schemes that use *multimethods*.

These are also very powerful techniques, but unfortunately are a little far outside of this introductory treatment of Clojure.

10.7 Summary

As a language, Clojure is arguably the most different from Java of the languages we've looked at. Its Lisp heritage, emphasis on immutability, and different approaches seem to make it into an entirely separate language. But its tight integration with the JVM, alignment of its type system (even when it provides alternatives, such as seqs), and the power of exploratory programming make it a very complementary language to Java.

The differences between the languages we've studied in this part clearly show the power of the Java platform to evolve, and to continue to be a viable destination for application development. This is also a testament to the flexibility and capability of the JVM.

- Clojure is dynamically typed and Java programmers need to be careful of runtime exceptions
- Exploratory and REPL-based development is a different feel to a Java IDE
- Clojure provides and promotes a very immutable style of programming
- Functional programming pervades Clojure - far more so than Java or Kotlin
- Seqs are a functional equivalent to Java's iterators and collections
- Macros define a compile-time transformation of Clojure source

11

Building with Gradle & Maven

This chapter covers

- Why build tools matter for a well-grounded developer
- Maven
- Gradle

11.1 Why build tools matter for a well-grounded developer

The JDK ships with a compiler to turn Java source code into class files as we saw in Chapter 4. Despite that fact, few projects of any size rely just on `javac`. Build tools are the norm for a number of reasons:

- Automating tedious operations
- Managing dependencies
- Ensuring consistency between developers

Although many options exist, two choices dominate the landscape today: Maven and Gradle. Understanding what these tools aim to solve, digging below the surface of how they get their job done, and understanding the differences between them - and how to extend them - will pay off for the well-grounded developer.

11.1.1 Automating tedious operations

`javac` can turn any Java source file into a class file, but there's more to building a typical Java project than that. Just getting all the files properly listed to the compiler could be tedious in a large project if done by hand. Build tools provide defaults for finding code and let you easily configure if you have a non-standard layout instead.

The default layout popularized by Maven and used by Gradle as well looks like this:

```

src
  main
    java
      com
        wellgrounded
          Main.java
  test
    java
      com
        wellgrounded
          MainTest.java

```

①
②
③

- ① main and test separate our production code from our test code
- ② Multiple languages easily coexist within one project with this structure
- ③ Further directory structure typically mirrors your package hierarchy

As you can see, testing is baked all the way into the layout of our code. Java's come a long way since the times when folks used to ask whether they really needed to write tests for their code. The build tools have been a key part in making testing available in a consistent manner everywhere.

NOTE

You probably already know about how to unit test in Java with JUnit or another library. We will discuss other forms of testing in Chapter 13.

While compiling to class files is the start of a Java program's existence, generally it isn't the end of the line. Fortunately, build tools also provide support for packaging your class files up into a JAR or other format for easier distribution.

11.1.2 Managing dependencies

In the early days of Java if you wanted to use a library, you had to find its JAR somewhere, download the file, and put it into the classpath for your application. This caused several problems - in particular the lack of a central, authoritative source for all libraries meant that a treasure hunt was sometimes necessary to find the JARs for less-common dependencies.

That obviously wasn't ideal, and so Maven (among other projects) gave the Java ecosystem repositories where tools could find and install dependencies for us. Maven Central remains to this day one of the most commonly used registries for Java dependencies on the internet.

Downloading all that code can be time-consuming too, so build tools have standardized on a few ways of reducing the pain by sharing artifacts between projects. With a local repository to cache, if a second project needs the same library it doesn't need to be downloaded again. This approach also saves disk space, of course, but the single source of artifacts is the real win here.

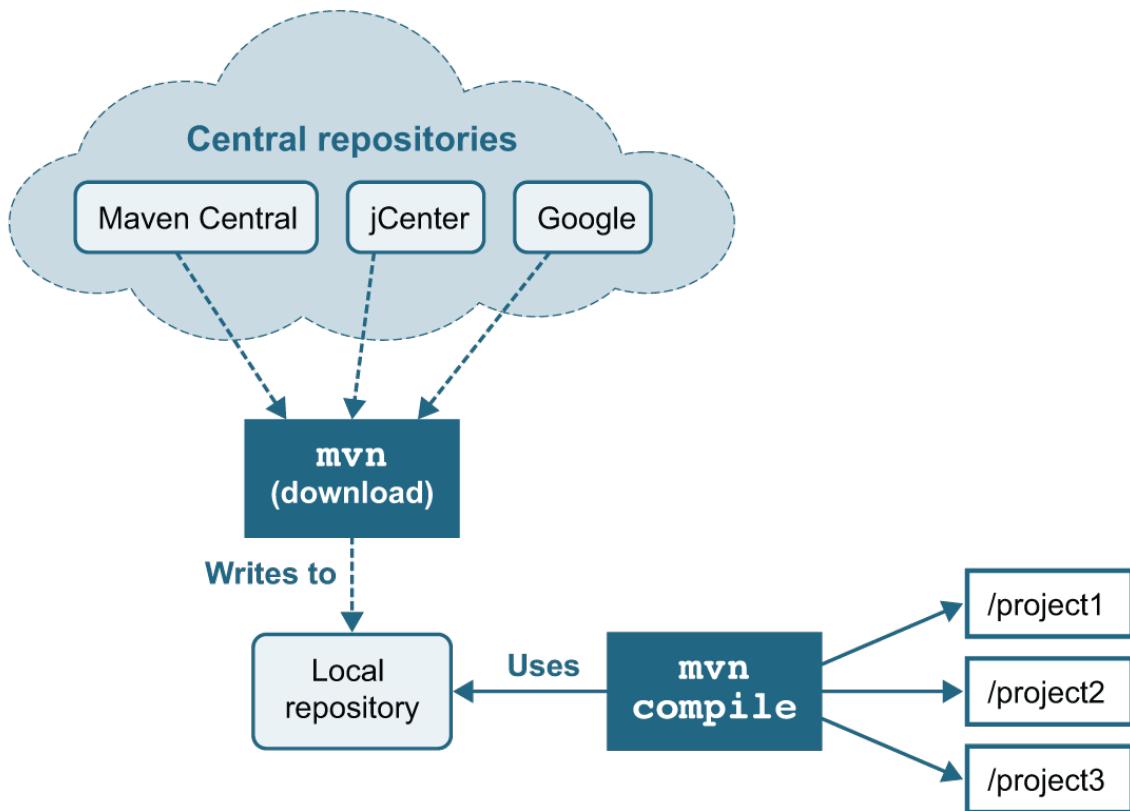


Figure 11.1 Maven’s local repository helping not only find dependencies online but manage them efficiently locally

NOTE

You might be wondering where modules fit in this dependency landscape. Modularized libraries are shipped as JAR files with the addition of the `module-info.class` file as we saw in Chapter 2. A modularized JAR can be downloaded from the standard repositories. The real differences come into play when you start compiling and running with modules - not in the packaging and distribution.

More than just providing a central place to find and download dependencies, though, registries opened the door for better management of *transitive dependencies*.

In Java, we commonly see this situation when a library that our project uses itself depends on *another* library. We actually already met transitive dependency of modules in Chapter 2, but the problem existed long before Java modules. In fact, before modules the problem was significantly worse.

Recall that JAR files are just zips - they don’t have any metadata that describes the dependencies of the JAR. This means that the dependencies of a JAR are just the union of all of the dependencies of all the classes in the JAR.

To make matters worse, the classfile format does not describe which version of a class is needed to satisfy the dependency - all we have is a symbolic descriptor of the class or method name that the class requires in order to link (as we saw in Chapter 3).

This implies two things:

1. An external source of dependency information is required
2. As projects get larger, the transitive dependency graph will get increasingly complex

With the explosion of open source libraries and frameworks to support developers the typical tree of transitive dependencies in a real project has only gotten larger and larger.

One potential bit of good news is that the situation for the JVM ecosystem is somewhat better than it is for, say, JavaScript. JavaScript lacks a rich, central runtime library which is guaranteed always to be present. This means that a lot of basic capabilities have to be managed as external dependencies. This introduces problems such as multiple incompatible libraries that each provide a version of a common feature and a fragile ecosystem where mistakes and hostile attacks can have a disproportionate impact on the commons. (E.g. the "left-pad" incident from 2016).

Java on the other hand, has a runtime library (the JRE) that contains a lot of commonly needed classes - and this is available in every Java environment. However, a real production application will require capabilities beyond those in the JRE - and will almost always have too many layers of dependencies to comfortably manage manually. The only solution is to automate.

A CONFLICT EMERGES

This automation is a boon for developers building on the rich ecosystem of open source code available, but upgrading dependencies often reveals problems as well. For instances, here's a dependency tree which might set us up for trouble:

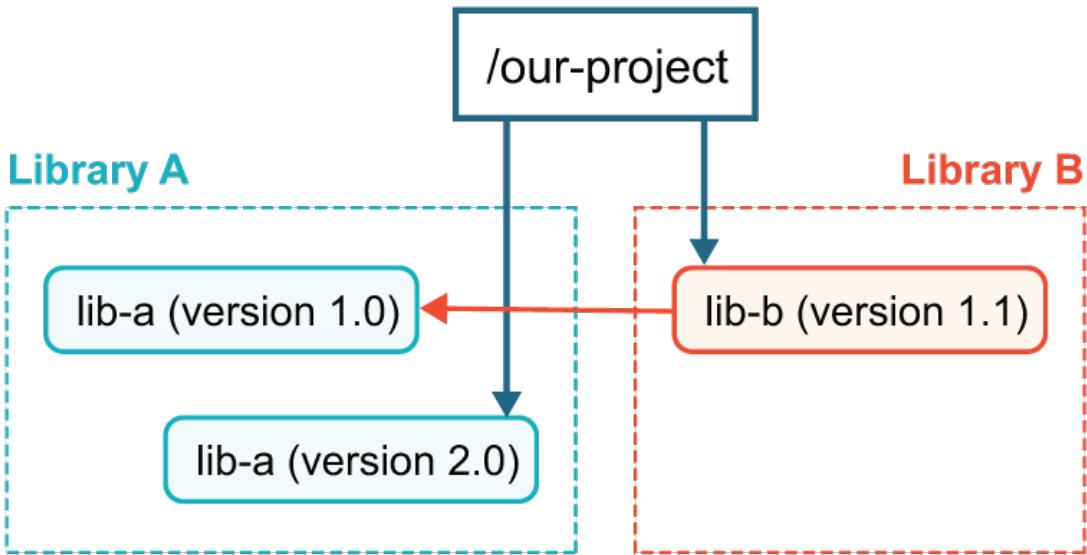


Figure 11.2 Conflicting transitive dependencies

We've asked explicitly for version 2.0 of `lib-a`, but our dependency `lib-b` has asked for the older version 1.0. This is known as a *dependency conflict* and depending on how it is resolved, it can cause a variety of other problems.

What types of breakage can result from mismatched library versions? This depends on the nature of the changes between the versions. Changes fall into a few categories:

- Stable APIs where only behavior changes between versions
- Added APIs where new classes or methods appear between versions
- Changed APIs where method signatures or interfaces extended change between versions
- Removed APIs where classes or methods are removed between versions

In the case of a) or b), you may not even notice which version of the dependency your build tool has chosen.

The most common case of c) is a change to the signature of a method between library versions. In our example above, if `lib-a` 2.0 altered the signature of a method that `lib-b` relied upon, when `lib-b` tried to call that method it would receive a `NoSuchMethodError` exception.

Removed methods in case d) would result in the same sorts of `NoSuchMethodError`. This includes "renaming" a method, which at the bytecode level isn't any different from removing a method and adding a new one that just happens to have the same implementation.

Classes are also prone to d) on deletion or renaming, and will cause a `NoClassDefFoundError`. It's also possible removal of interfaces from a class could land you with an ugly `ClassCastException`.

This list of issues with conflicting transitive dependencies is by no means exhaustive. It all boils down to *what* actually changes between two versions of the same package.

In fact, communicating about the nature of changes between versions is a common problem across languages. One of the most broadly adopted approaches to the handling the problem is *Semantic Versioning*. Semantic versioning gives us a vocabulary for stating requirements of our transitive dependencies, which in turn allows the machines to help us sorting them out.

When using semantic versioning:

- *MAJOR* version increments (1.x 2.x) on breaking changes to your API, like cases c) and d) above
- *MINOR* version increments (1.1 1.2) on backwards compatible additions like case b)
- *PATCH* increments on bug fixes (1.1.0 1.1.1).

While not foolproof, it at least gives expectation to what level of changes come with a version update, and is broadly used in open source.

Having gotten a taste of why dependency management isn't easy, rest assured that both Maven and Gradle provide tooling to help. Later in the chapter we'll look in detail at what each tool provides to unravel problems when you hit dependency conflicts.

11.1.3 Ensuring consistency between developers

As projects grow in volume of code and developers involved, they often get more complex and harder to work with. Your build tooling can lessen this pain, though. Built-in features like ensuring everyone is compiling and running the same tests are a start. But there's many additions beyond the basics to consider as well.

Tests are good, but how certain are you that *all* your code is tested? Code coverage tools are key for detecting what code is hit by your tests and what isn't. While arguments swirl on the internet about the right target for code coverage, the line-level output coverage tools provide can save you from missing a test for that one extra special conditional.

Java as a language also lends itself well to a variety of static analysis tools. From detecting common patterns (i.e. overriding `equals` without overriding `hashCode`) to sniffing out unused variables, static-analysis lets a computer validate aspects of the code that are legal but will bite you in production.

Beyond the realms of correctness, though, there are style and formatting tools. Ever fought with someone about where the curly braces should go in a statement? How to indent your code? Agreeing once to a set of rules, even if they aren't all perfectly to your taste, lets you focus forever after in the project on the actual work instead of nitpicking details about how the code looks.

Last and certainly not least, your build tool is a pivotal central point for providing custom functionality. Are there special setup or operational commands folks need to run periodically for your project? Validations your project should run after a build but before you deploy? All of these are excellent to consider wiring into the build tooling so they're available to everyone working with the code. Both Maven and Gradle provide many ways to extend them for your own logic and needs.

Hopefully you're now convinced that build tools aren't just something to set up once on a project, but worth investment in understanding. Let's start by taking a look at one of the most common: Maven.

11.2 Maven

Early in Java history, the Ant framework was the default build tool. With tasks described in XML, it allowed a more Java-centric way to script builds than tools like CMake. But Ant lacked structure around how to configure your build - what the steps were, how they related, how dependencies were managed.

Maven addressed many of these gaps with its concept of a standardized *build lifecycle* and a consistent approach to handling dependencies.

11.2.1 The build lifecycle

Maven is an opinionated tool. One of the biggest areas where these opinions show is in its build lifecycles. Rather than users defining their own tasks and determining their order, Maven has a *default lifecycle* encompassing the usual steps, known as *phases*, that you'd expect in a build.

While not comprehensive, the following phases capture the high points in the default lifecycle.

- validate - check project configuration is correct and can build
- compile - compile the source code
- test - run unit tests
- package - generate artifacts such as JAR files
- verify - run integration tests
- install - install package to local repository
- deploy - make package result available to others, typically run from CI environment

Chances are that these map to most of the steps you'll take from source code to a deployed application or library. This is a major bonus to Maven's opinionated approach - any Maven project will share this same lifecycle. Your knowledge of how to run builds is more transportable than it used to be.

While the phases are well-defined in Maven, every project needs something special in the details.

In Maven's model, various *plugins* attach *goals* to these phases. A goal is a concrete task, with the implementation of how to execute it.

Beyond the default lifecycle, Maven also includes the *clean* and *site* lifecycles. The *clean* lifecycle is intended for clean-up (e.g. removing intermediate build results), while *site* is meant for documentation generation.

We'll look closer at hooking into a lifecycle later when we discuss extending Maven, but if you truly need to redefine the universe Maven does support authoring fully custom lifecycles. This is a very advanced topic, however and beyond the scope of this book.

11.2.2 Commands/POM intro

Maven is a project of the Apache Software Foundation and is open-source. Installation instructions can be [found on the project website](#).

Typically, Maven is installed globally on a developer's workstation and it works on any not-ancient JVM (JDK 7 or higher). Once installed, invoking it gets us this output:

```
~: mvn

[INFO] Scanning for projects...
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time:  0.066 s
[INFO] Finished at: 2020-07-05T21:28:22+02:00
[INFO] -----
[ERROR] No goals have been specified for this build. You must specify a
valid lifecycle phase or a goal in the format <plugin-prefix>:<goal> or
<plugin-group-id>:<plugin-artifact-id>[:<plugin-version>]:<goal>.
Available lifecycle phases are: validate, initialize, ....
```

Of particular interest is the message that `No goals have been specified for this build`. This indicates that Maven doesn't know anything about our project. We provide that information by the `pom.xml` file, which is the center of the universe for a Maven project.

While a full-blown `pom.xml` file can be intimidatingly long and complex, you can get started with much less. For example, a more-or-less minimal `pom.xml` file looks like this:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.wellgrounded</groupId>
  <artifactId>example</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>example</name>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>
</project>
```

① ②

- ① Identifying our project
- ② The Maven plugins default to Java 1.6. We obviously want a newer version.

Our `pom.xml` file declares two particularly important fields - the `groupId` and the `artifactId`. These fields combine with a version to form the *GAV coordinates* (group, artifact, version) which uniquely, globally identifies a specific release of your package. `groupId` typically specifies the company, organization, or open source project responsible for the library, while `artifactId` is the name for the specific library. GAV coordinates are often expressed with each part separated by a colon (:), such as `org.apache.commons:collections4:4.4` or `com.google.guava:guava:30.1-jra`.

These coordinates are important not just for configuring your project locally. Coordinates act as the address for dependencies so our build tooling can find them. The following sections will dig into the mechanics of how we express those dependencies in more detail.

Much like Maven standardized the build lifecycle, it also popularized the standard layout we saw earlier in section 11.1.1. If you follow these conventions, you don't have to tell Maven anything about your project for it to be able to compile.

```
.
pom.xml
src
  main
    java
      com
        wellgrounded
          Main.java
  test
    java
      com
        wellgrounded
          MainTest.java
```

Notice the parallel structures - `src/main/java` and `src/test/java` - with the same directories mapping to our package hierarchy. This convention keeps the test code separate from the main app code, which simplifies the process to package our main code for deployment, excluding the test code which users of a package won't typically want or use.

While you're getting used to Maven it's a good idea to stick to the conventions, standard layouts, and other defaults that Maven provides. As we mentioned, it's an opinionated tool, so it's better to stay within the guardrails it provides while you're learning. Experienced Maven developers can (and do) stray outside the conventions and break the rules, but let's not try to run before we walk.

11.2.3 Building

We saw previously that just running `mvn` on the command-line warns us that we need to choose a *lifecycle phase* or *goal* to actually take action. Most often we'll want to run a phase, which may include many goals.

The simplest place to get started is compiling our code by requesting the `compile` phase.

```
~: mvn compile

[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.wellgrounded:example >-----
[INFO] Building example 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] -- maven-resources-plugin:2.6:resources (default-resources) ①
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] ----- maven-compiler-plugin:3.1:compile (default-compile) -- ②
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to ./maven-example/target/classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  0.940 s
[INFO] Finished at: 2020-07-05T21:46:25+02:00
[INFO] -----
```

- ① Although we don't have resources in our project, the `maven-resources-plugin` from the default lifecycle checks for us
- ② Our actual compilation is provided by `maven-compiler-plugin`

Maven defaults our output to the `target` directory. After our `mvn compile` we can find the class files under `target/classes`. Close inspection will reveal we only built the code under our `main` directory. If we want to compile our tests you can use the `test-compile` phase.

The default lifecycle includes more than just compilation. For instance, `mvn package` for the project above will result in a JAR file at `target/example-1.0-SNAPSHOT.jar`.

While we can use this JAR as a library, if we try to run it via `java -jar target/example-1.0-SNAPSHOT.jar`, we'll find that Java complains it can't find a main class. To see how we start growing our Maven build, let's change it so the produced JAR is a runnable application.

11.2.4 Controlling the manifest

The JAR Maven produced from `mvn package` was missing a *manifest* to tell the JVM where to look for a `main` method on startup. Fortunately, Maven ships with a plugin for constructing JARs that knows how to write the manifest. The plugin exposes configuration via our `pom.xml` after the `properties` element and still inside the `project` element.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <mainClass>com.wellgrounded.Main</mainClass>
            <Automatic-Module-Name>
              com.wellgrounded
            </Automatic-Module-Name>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- ➊ `maven-jar-plugin` is the plugin name. You can spot this easily in the output when running the `mvn package` command.
- ➋ Each plugin has its own specialized `configuration` element with different child elements and attributes supported.
- ➌ `<manifest>` configures the resulting JAR's manifest contents.
- ➍ Configuring our automatic module name.

Adding this section sets up the main class so the `java` launcher knows how to directly execute the JAR. We have also added an automatic module name - this is to be good citizens in the modular world. As we discussed back in Chapter 2, even if the code we're writing is not modular (as in this case), it still makes sense to provide an explicit automatic module name so modular applications can more easily use our code.

This pattern of setting configuration under a `plugin` element is very standard in Maven. To simplify things, most default plugins will kindly warn if you use an unsupported or unexpected configuration property, although the details may vary by plugin.

11.2.5 Adding another language

As we discussed in Chapter 8, an advantage of the JVM as a platform is the ability to use multiple languages within the same project. This may be useful when a specific language has better facilities for a given part of your application, or even to allow gradual conversion of an application from one language to another.

Let's take a look at how we would configure our simple Maven project to build some classes from Kotlin instead of Java.

Our standard layout is fortunately already set to allow for easy adding languages.

```
.
  pom.xml
  src
    main
      java
        com
          wellgrounded
            Main.java
    kotlin
      com
        wellgrounded
          MessageFromKotlin.kt
  test
    java
      com
        wellgrounded
          MainTest.java
```

- ➊ We keep our `kotlin` code in its own subdirectory so it's easy to tell what paths use which compiler to produce class files.
- ➋ Packages can mix between the languages, as the resulting class files don't have direct knowledge of what language they were generated from.

Unlike Java, Maven by default doesn't know how to compile Kotlin so we need to add `kotlin-maven-plugin` in our `pom.xml`. We recommend consulting the [Kotlin documentation](#) for the most up to date usage, but we'll demonstrate here so you can know what to expect.

If a project is fully written in Kotlin, compilation only needs the plugin added and attached to the `compile` goal.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-plugin</artifactId>
      <version>1.4.31</version>          ①
      <executions>
        <execution>
          <id>compile</id>
          <goals>          ②
            <goal>compile</goal>
          </goals>
        </execution>
        <execution>
          <id>test-compile</id>
          <goals>
            <goal>test-compile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

- ① Current version of Kotlin as of when this chapter was written.
- ② Add this plugin to the goals for compiling main and test code.

The situation gets more complex when mixing Kotlin and Java. Mavens' default maven-compiler-plugin which compiles Java for us needs to be overridden to let Kotlin compile first, or our Java code will be unable to use the Kotlin classes.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-maven-plugin</artifactId>
      <version>1.4.31</version>
      <executions>
        <execution>
          <id>compile</id>
          <goals>
            <goal>compile</goal>
          </goals>
          <configuration>
            <sourceDirs>
              <sourceDir>${project.basedir}/src/main/kotlin</sourceDir>
              <sourceDir>${project.basedir}/src/main/java</sourceDir>
            </sourceDirs>
          </configuration>
        </execution>
        <execution>
          <id>test-compile</id>
          <goals>
            <goal>test-compile</goal>
          </goals>
          <configuration>
            <sourceDirs>
              <sourceDir>${project.basedir}/src/test/kotlin</sourceDir>
              <sourceDir>${project.basedir}/src/test/java</sourceDir>
            </sourceDirs>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.5.1</version>
      <executions>
        <execution>
          <id>default-compile</id>
          <phase>none</phase>
        </execution>
        <execution>
          <id>default-testCompile</id>
          <phase>none</phase>
        </execution>
        <execution>
          <id>java-compile</id>
          <phase>compile</phase>
          <goals>
            <goal>compile</goal>
          </goals>
        </execution>
        <execution>
          <id>java-test-compile</id>
          <phase>test-compile</phase>
          <goals>
            <goal>testCompile</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

- ① Add the `kotlin-maven-plugin` mostly as before, making sure now it's aware of both Java and Kotlin paths.
- ② The Kotlin compiler needs to know about both our Kotlin and Java code locations.
- ③ Disable the `maven-compiler-plugin` defaults for building Java as these force it to run first.
- ④ Re-apply the `maven-compiler-plugin` to the `compile` and `test-compile` phases. These will now be added after the `kotlin-maven-plugin`.

Your project will need a dependency on at least the Kotlin standard library, so we add that explicitly.

```
<dependencies>
  <dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-stdlib</artifactId>
    <version>1.4.31</version>
  </dependency>
</dependencies>
```

With this in place, our multi-lingual project builds and runs as before.

11.2.6 Testing

Once your code builds, a smart next step is to test it. Maven integrates testing deeply into its lifecycle. In fact, where compilation of your main code is only a single phase, Maven supports two separate phases of testing out of the box - `test` and `integration-test`.

`test` is used for typical unit testing, while the `integration-test` phase runs after construction of artifacts such as JARs, with the intent of performing end-to-end validation on your final outputs.

NOTE

Integration tests may also be run with JUnit because, despite the name, JUnit is a very capable test runner for more than just unit testing. Do not fall into the trap of thinking that any test executed by JUnit is automatically a unit test! We'll examine the different types of tests in detail in Chapter 13.

Almost any project will benefit from some testing. As you might expect from Maven's opinionated stance, testing happens (by default) using the near ubiquitous framework JUnit. Other frameworks are just a plugin away.

While the standard plugins know about running JUnit, we still must declare the library as a dependency so Maven knows how to compile our tests. You can add a library with a snippet like the following under the `<project>` element.

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

①

- ① `<scope>` indicates this library is only needed for the `test-compile` phase

With that in place, we can run our unit tests.

```
~:mvn test

[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.wellgrounded:example >-----
[INFO] Building example 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] .....
[INFO]
[INFO] -- maven-surefire-plugin:2.12.4:test (default-test) @ example - ①
[INFO] Surefire report dir: ./maven-example/target/surefire-reports

-----
T E S T S
-----
Running com.wellgrounded.MainTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.010 s
[INFO] Finished at: 2020-07-06T15:45:22+02:00
[INFO] -----
```

- ① Maven's default for running JUnit tests is the `maven-surefire-plugin`

By default, the Surefire plugin runs all unit tests in the standard location `src/test/*` during the `test` phase. If we want to take advantage of the `integration-test` phase, it's recommended to use a separate plugin, such as `maven-failsafe-plugin`. Failsafe is maintained by the same folks who make `maven-surefire-plugin`, and specifically targets the integration testing case. We add the plugin in our the `<build><plugins>` section we previously used for configuring our manifest.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.0.0-M5</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Failsafe treats the following filename patterns as integration tests, although that can be configured.

- `**/IT*.java`
- `**/*IT.java`
- `**/*ITCase.java`

Since it's part of the same suite of plugins, Surefire is also aware of this convention and excludes these tests from the `test` phase.

It's recommended to run integration tests via `mvn verify` rather than `mvn integration-test`. `verify` includes `post-integration-test` for cleaning up after long-running tests, while `integration-test` doesn't.

```
~: mvn verify

[INFO] ... compilation output omitted for length ...

[INFO] --- maven-failsafe-plugin:3.0.0-M5:integration-test @ example ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.wellgrounded.LongRunningIT
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,
[INFO] Time elapsed: 0.032 s - in com.wellgrounded.LongRunningIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M5:verify (default) @ example ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

11.2.7 Dependency management

A key feature Maven brought to the ecosystem was a standard format for expressing dependency management information via the `pom.xml` file. Maven also established a central repository for libraries. Maven can walk your `pom.xml` and the `pom.xml` files from your dependencies to determine the entire set of *transitive dependencies* your application requires.

The process of walking the tree and finding all the necessary libraries is called *dependency resolution*. While critical for managing modern applications, the process does have its sharp edges.

To see where the problems arise, let's revisit the project setup we saw earlier in section 11.1.2. Recall that the project's dependencies have resulted in a tree that looks like this:

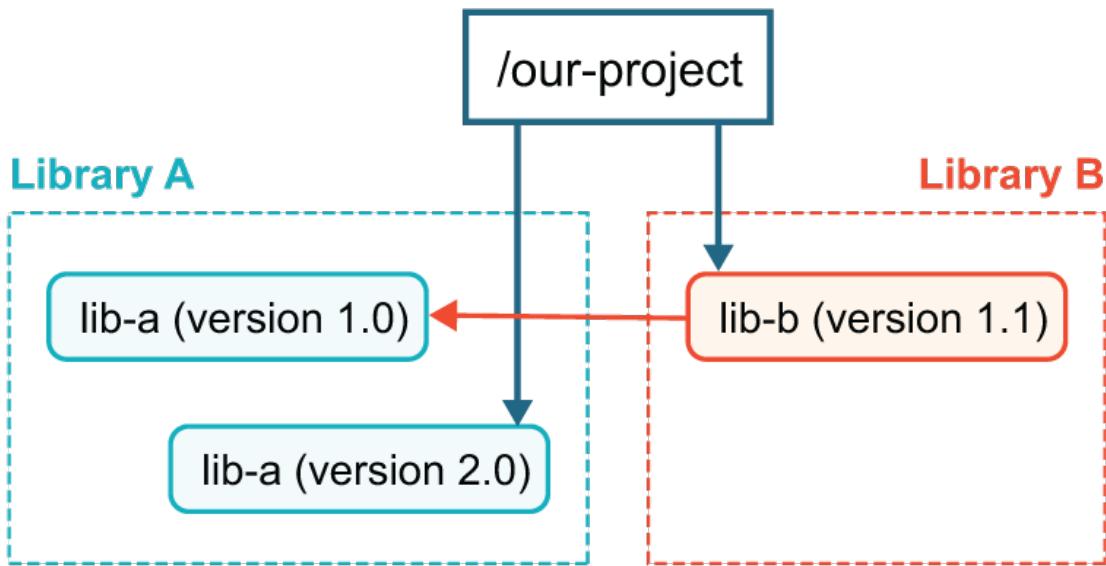


Figure 11.3 Conflicting transitive dependencies where dependency requests an older version

Here we've asked explicitly for version 2.0 of `lib-a`, but our dependency `lib-b` has asked for the older version 1.0.

Maven's dependency resolution algorithm favors the version of a library closest to the root. The end result of the configuration above is that we will use `lib-a` 2.0 in our application. As we outlined in section 11.1.2, this may work fine or be disastrously broken.

Another common scenario that can also cause problems is when the reverse occurs and the dependency that is closest to the root is *older* than the one expected as a transitive dependency:

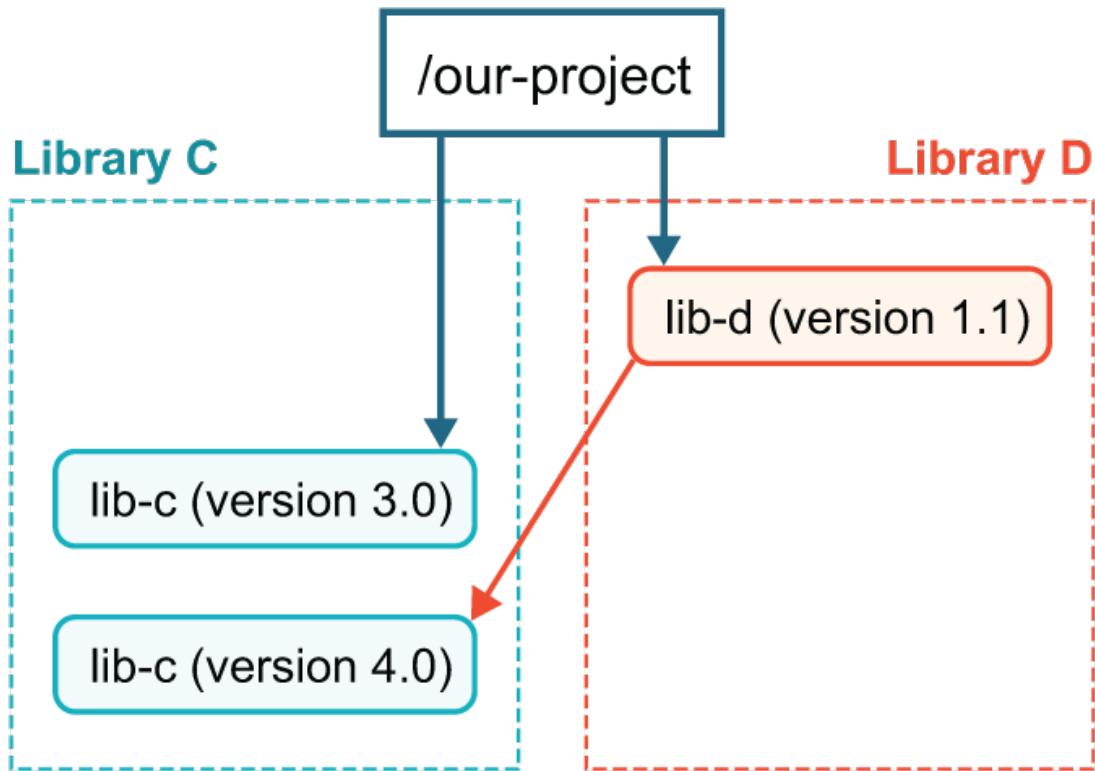


Figure 11.4 Conflicting transitive dependencies where dependency asks for a newer version

In this case, it's entirely possible that `lib-d` is relying on an API in `lib-c` that didn't exist in version 3.0 - and so adding a dependency on `lib-d` to a project that is already using `lib-c` will result in runtime exceptions.

NOTE

Given those possibilities, we recommend any package your code directly interacts with should be declared explicitly in your `pom.xml`. If you don't, and instead rely upon transitive dependency then updating your direct dependency could result in unexpected build breakage.

Before we can solve our dependency problems, it's important to know what our dependencies are. Maven has us covered with the `mvn dependency:tree` command.

```

~:mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.wellgrounded:example >-----
[INFO] Building example 1.0-SNAPSHOT
[INFO] ----- [ jar ] -----
[INFO]
[INFO] -- maven-dependency-plugin:2.8:tree (default-cli) @ example --
[INFO] com.wellgrounded:example:jar:1.0-SNAPSHOT
[INFO] \- junit:junit:jar:4.12:test
[INFO]     \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 0.790 s
[INFO] Finished at: 2020-08-13T23:02:10+02:00
[INFO] -----

```

①

②

- ① Our direct dependency on JUnit from the `pom.xml` file
- ② JUnit has a transitive dependency on `hamcrest-core`

Let's imagine we want to use an additional testing library, but that library brings along a dependency on `hamcrest-core` at a different version than what `junit` is asking for - say 2.2 .

The best possible approach is finding newer versions of our dependencies that can all agree on their dependencies. In fact, JUnit 5 changes its entire relationship to Hamcrest for assertions, but sadly the jump from JUnit 4 to 5 isn't as simple as bumping a version in your `pom.xml`. In acceptable SemVer form, the jump from 4 to 5 introduced breaking changes that almost certainly require code changes to accommodate.

This leaves us looking for other tools to deal with the conflict. Two main approaches come into play if we can't find a natural resolution. Be aware that both of these solutions require finding *some* compatible version that will satisfy your dependencies.

If one of your dependencies specifies a version that everyone could agree on, but it isn't being chosen by Maven's resolution algorithm, you can tell Maven to exclude parts of the tree when resolving. If our helper library uses a newer `hamcrest-core` that happens to work fine with JUnit, we can just ignore what JUnit asks for.

```

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
    <exclusions>
      <exclusion>
        <groupId>org.hamcrest</groupId>
        <artifactId>hamcrest-core</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>com.wellgrounded</groupId> ①
    <artifactId>testhelpers</artifactId>
    <version>0.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>

```

- ① Exclude junit version of hamcrest-core
- ② Transitive dependency from com.wellgrounded.testhelpers proceed normally.

In the worst case, perhaps neither library expresses the compatible version. To handle this, we specify the precise version as a direct dependency in our project. By its resolution rules, Maven will choose that version as it is closer to the project root. While this convinces the tool to do what we want, we are taking on the risk of runtime errors from mixing libraries versions, so it is important to test the interactions thoroughly.

```

<dependencies>
  <dependency>
    <groupId>junit</groupId> ①
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.wellgrounded</groupId> ①
    <artifactId>testhelpers</artifactId>
    <version>0.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.hamcrest</groupId> ②
    <artifactId>hamcrest-core</artifactId>
    <version>2.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>

```

- ① Our dependencies pulls in hamcrest-core at a different version
- ② Forcing resolution on hamcrest-core to the version we want

11.2.8 Reviewing

Our build process is an excellent spot to hook in additional tooling and checks. One key bit of information is code coverage which informs us what parts of our code our tests execute.

A leading option for code coverage in the Java ecosystem is [JaCoCo](#). JaCoCo can be configured to enforce certain coverage levels during testing and will output reports that tell you what is and isn't covered.

Enabling JaCoCo requires only adding a plugin to the `<build><plugins>` section of your `pom.xml` file. It doesn't enable itself by default, so you have to tell it when it should execute. In this example we've bound it to the `test` phase:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.5</version>
      <executions>
        <execution>①
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>②
          <id>report</id>
          <phase>test</phase>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

- ① JaCoCo needs to start running early in the process. This adds it to the `initialize` phase
- ② Tell JaCoCo to `report` during the `test` phase

This produces reports on all of your classes in `target/site/jacoco` by default, with a full HTML version at `index.html` to be explored.

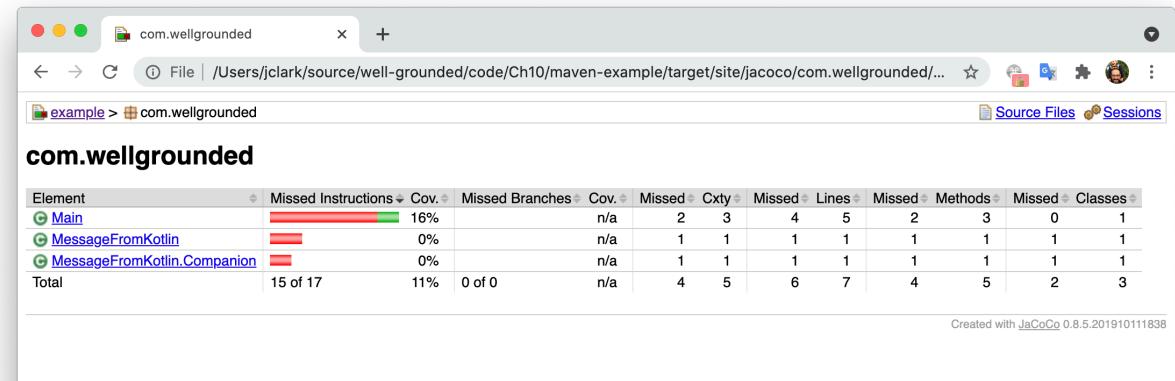


Figure 11.5 JaCoCo coverage report page

11.2.9 Moving beyond Java 8

In Chapter 1, we noted a series of modules that belonged with Java Enterprise Edition, but were present in the core JDK. These were deprecated with JDK 9 and removed with JDK 11 but remain available as external libraries.

- `java.activation` (JAF)
- `java.corba` (CORBA)
- `java.transaction` (JTA)
- `java.xml.bind` (JAXB)
- `java.xml.ws` (JAX-WS, plus some related technologies)
- `java.xml.ws.annotation` (Common Annotations)

If your project relies on any of these modules, your build might break when you move to a more recent JDK. Fortunately a few simple dependency additions in your `pom.xml` address the issue.

```

<dependencies>
  <dependency>
    <groupId>com.sun.activation</groupId>          ①
    <artifactId>jakarta.activation</artifactId>
    <version>1.2.2</version>
  </dependency>
  <dependency>
    <groupId>org.glassfish.corba</groupId>           ②
    <artifactId>glassfish-corba-omgapi</artifactId>
    <version>4.2.1</version>
  </dependency>
  <dependency>
    <groupId>javax.transaction</groupId>               ③
    <artifactId>javax.transaction-api</artifactId>
    <version>1.3</version>
  </dependency>
  <dependency>
    <groupId>jakarta.xml.bind</groupId>                 ④
    <artifactId>jakarta.xml.bind-api</artifactId>
    <version>2.3.3</version>
  </dependency>
  <dependency>
    <groupId>jakarta.xml.ws</groupId>                  ⑤
    <artifactId>jakarta.xml.ws-api</artifactId>
    <version>2.3.3</version>
  </dependency>
  <dependency>
    <groupId>jakarta.annotation</groupId>                ⑥
    <artifactId>jakarta.annotation-api</artifactId>
    <version>1.3.5</version>
  </dependency>
</dependencies>

```

- ① java.activation (JAF)
- ② java.corba (CORBA)
- ③ java.transaction (JTA)
- ④ java.xml.bind (JAXB)
- ⑤ java.xml.ws (JAX-WS, plus some related technologies)
- ⑥ java.xml.ws.annotation (Common Annotations)

11.2.10 Multi-release JARs in Maven

A feature that arrived in JDK 9 was the ability to package JARs that target different code for different JDKs. This allows us to take advantage of new features in the platform, while still supporting clients of our code on older versions.

In Chapter 2 we examined the feature and hand-crafted the specific JAR format necessary to enable this capability. This is exactly the sort of tedious process that Maven was built to automate.

Although the output format in our JAR is all that really matters to enable the multi-release feature, we'll mimic the structure in our code layout for clarity. The code in `src` is the baseline functionality which will be seen by any JDK by default. The code under `versions` optionally

replaces specific classes with an alternate implementation.

```
.
  pom.xml
  src
    main
      java
        wgjd2ed
          GetPID.java
          Main.java
  versions
    11
      src
        wgjd2ed
          GetPID.java
```

Maven's defaults will find and compile our code in `src/main`, but we have two complications we need to sort out:

1. Maven needs to *also* find our code in the `versions` directory
2. Further, Maven needs to compile that source targeted to a different JDK than the main project

Both of these goals can be accomplished by configuring the `maven-compiler-plugin` that builds our Java class files. We introduce two separate `<execution>` steps—one to compile the base code targeting JDK 8, and then a second pass to compile the versioned code targeting JDK 11.

```

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <executions>
      <execution>
        <id>compile-java-8</id> ❶
        <goals>
          <goal>compile</goal>
        </goals>
        <configuration>
          <source>1.8</source> ❷
          <target>1.8</target>
        </configuration>
      </execution>
      <execution>
        <id>compile-java-11</id> ❸
        <phase>compile</phase>
        <goals>
          <goal>compile</goal>
        </goals>
        <configuration>
          <compileSourceRoots> ❹
            <compileSourceRoot>
              ${project.basedir}/versions/11/src
            </compileSourceRoot>
          </compileSourceRoots>
          <release>11</release> ❺
          <multiReleaseOutput>true</multiReleaseOutput>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>

```

- ❶ Execution step to compile for JDK 8.
- ❷ We'll run our build with JDK 11, so we explicitly target this step's output to JDK 8.
- ❸ Second execution step for targeting JDK 11.
- ❹ Tell Maven about our alternate location for the version-specific code.
- ❺ Setting `release` and `multiReleaseOutput` tells Maven which JDK this versioned code is intended for, and asks it to put at the correct multi-release location in output.

This gets our JAR built and packaged with the right layout. There's one more step, and that's marking the manifest as multi-release. This is configured in the `maven-jar-plugin`, close to where we made our application JAR executable in section 11.2.4.

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.2.0</version>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
        <mainClass>wgjd2ed.Main</mainClass>
      </manifest>
      <manifestEntries>
        <Multi-Release>true</Multi-Release> ①
      </manifestEntries>
    </archive>
  </configuration>
</plugin>

```

① Attribute to mark the JAR as multi-release

With that we can execute our code against different JDKs and see it behave as expected. In the case of our sample app, the base implementation for JDK 8 will output and additional version message so we can see it's working.

```

~:mvn clean compile package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< wgjd2ed:maven-multi-release >-----
[INFO] Building maven-multi-release 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] .... Lots of additional steps
[INFO]
[INFO] - maven-jar-plugin:3.2.0:jar (default-jar) @ maven-multi-release -
[INFO] Building jar: ~/target/maven-multi-release-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.813 s
[INFO] Finished at: 2021-03-05T09:39:16+01:00
[INFO] -----


~:java -version
openjdk version "11.0.6" 2020-01-14
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.6+10)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.6+10, mixed mode)

~:java -jar target/maven-multi-release-1.0-SNAPSHOT.jar
75891

# Change JDK versions by your favorite means.....

~:java -version
openjdk version "1.8.0_265"
OpenJDK Runtime Environment (AdoptOpenJDK)(build 1.8.0_265-b01)
OpenJDK 64-Bit Server VM (AdoptOpenJDK)(build 25.265-b01, mixed mode)

~:java -jar target/maven-multi-release-1.0-SNAPSHOT.jar
Java 8 version...
76087

```

The path to using new features in the JDK without abandoning older clients is all set!

11.2.11 Maven and modules

In Chapter 2 we examined the JDK’s new module system in detail. Let’s look at how it influences our build scripting. We’ll start looking at a simple library that exposes one of its packages publicly, while hiding the other.

A MODULAR LIBRARY

Modular projects vary slightly in their code layout from the strict Maven standard. The `main` directory instead reflects the name of the module.

```
.
  pom.xml
  src
    com.wellgrounded.modlib
      java
        com
          wellgrounded
            hidden
              CantTouchThis.java
            visible
              UseThis.java
```

- ➊ Our modular code directory
- ➋ A class we intend to keep private
- ➌ A class we intend to share publicly through our module

Having made that change, we have to inform Maven of this new location to look for source code to compile.

```
<build>
  <sourceDirectory>src/com.wellgrounded.modlib/java</sourceDirectory>
</build>
```

The final piece to making our library modular is the addition of a `module-info.java` at the root of our code (alongside the `com` directory). This will name our module, and declare what we allow access to.

```
module com.wellgrounded.modlib {
  exports com.wellgrounded.modlib.visible;
}
```

Everything else about this simple library remains the same, and if we `mvn package` we’ll get a JAR file in `target`. Before we proceed further, we can also put this library into the local Maven cache via `mvn install`

NOTE

The JDK’s module system is about access control at build and runtime, not packaging. A modular library can be shared as a plain old JAR file, just with the additional `module-info.class` included to tell modular applications how to interact with it.

Now that we have a modular library, let's build a modular application to consume it.

A MODULAR APPLICATION

Our modular application gets a similar layout to what we used for the library.

```
.
  pom.xml
  src
    com.wellgrounded.modapp
      java
        com
          wellgrounded
            Main.java
      module-info.java
```

Our `module-info.java` for the application declares our name, and states that we require the package exported by our library.

```
module com.wellgrounded.modapp {
    requires com.wellgrounded.modlib;
}
```

This by itself doesn't tell Maven where to find our library JAR, though, so we include it as a normal `<dependency>`.

```
<dependencies>
  <dependency>
    <groupId>com.wellgrounded</groupId>      ①
    <artifactId>modlib</artifactId>
    <version>2.0</version>
  </dependency>
</dependencies>
```

- ① Our library from the prior section, installed into the local Maven repository.

When we're compiling and subsequently running, it's important that this dependency be placed on the module path instead of the class path. How does Maven accomplish this? Fortunately, recent versions of the `maven-compiler-plugin` are smart enough to notice that 1) our application has a `module-info.java` so it's modular, and 2) the dependency includes `module-info.class` so it too is a module. As long as you are on a recent version of `maven-compiler-plugin` (3.8 worked great at the time of writing), Maven figures it out for you.

Our application code is perfectly normal Java, and we can use the modular library's functionality as intended.

```
package com.wellgrounded.modapp;

import com.wellgrounded.modlib.visible.UseThis; ①

public class Main {
    public static void main(String[] args) {
        System.out.println(UseThis.getMessage()); ②
    }
}
```

- ① import from the module just like any other package
- ② Use the class from our module to get a message

You may remember that we had another package in our library that we didn't provide access to. What happens if we modify our application to try and pull that in?

```
package com.wellgrounded.modapp;

import com.wellgrounded.modlib.visible.UseThis;
import com.wellgrounded.modlib.hidden.CantTouchThis; ①

public class Main {
    public static void main(String[] args) {
        System.out.println(UseThis.getMessage());
    }
}
```

- ① com.wellgrounded.modlib.hidden was not listed in the library's exports

Compiling this will give us an error straight away.

```
[INFO] - maven-compiler-plugin:3.8.1:compile @ modapp ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 2 source files to /mod-app/target/classes
[INFO] -----
[ERROR] COMPILATION ERROR :
[INFO] -----
[ERROR]
src/com/wellgrounded/modapp/java/com/wellgrounded/Main.java:[4,31] ①
  package com.wellgrounded.modlib.hidden is not visible (package
  com.wellgrounded.modlib.hidden is declared in module
  com.wellgrounded.modlib, which does not export it)

[INFO] 1 error
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
```

- ① javac and the module system won't even let us try to touch things that aren't exported!

Maven's tooling has come a long way since the release of modules in JDK 9. All the standard scenarios are well covered with a minimum of additional configuration required.

Before we go, though, let's take one brief tangent. Throughout this section, `module-info.class` was frequently the signal to Maven that it should start applying modular rules. But modules are

an *opt-in* feature in the JDK to preserve compatibility with the vast quantities of pre-modular code out there.

What happens if we build the same application using our modular library, but the application doesn't mark itself to use modules by including the `module-info.java` file? In that case, the library - although it is modular - will be included via the classpath. This places it in the unnamed module along with the application's own code, and all those access restrictions we defined in the library are effectively ignore.

A sample application is included in the supplement alongside the modular one that uses our library by classpath so you can see more clearly how opting into or out of modules works.

With that our tour of Maven's default features is done. But what do we do if we need to extend the system beyond the admittedly vast array of plugins we can find online?

11.2.12 Authoring Maven plugins

Even the most basic defaults in Maven are supplied as plugins, and there's no reason you can't write one too when we need to do more.

As we've seen, referencing a plugin is a lot like pulling in a dependent library. It isn't surprising, then, that we implement our Maven plugins as separate JAR files.

For our example, we start with a `pom.xml` file. Much of the boilerplate is similar to before with a couple small additions.

```

<project>
  <modelVersion>4.0.0</modelVersion>

  <name>A Well-Grounded Maven Plugin</name>
  <groupId>com.wellgrounded</groupId>
  <artifactId>wellgrounded-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging> ①

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies> ②
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>3.0</version>
    </dependency>

    <dependency>
      <groupId>org.apache.maven.plugin-tools</groupId>
      <artifactId>maven-plugin-annotations</artifactId>
      <version>3.4</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>

```

- ① Let Maven know we intend to build a plugin package.
- ② Maven API dependencies our implementation will need.

That gets us set to start adding code. Placing a Java file in the standard layout location, we implement what is called a **Mojo** - effectively a Maven *goal*:

```

package com.wellgrounded;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugins.annotations.Mojo;

@Mojo(name = "wellgrounded")
public class WellGroundedMojo extends AbstractMojo
{
    public void execute() throws MojoExecutionException
    {
        getLog().info("Extending Maven for fun and profit.");
    }
}

```

Our class extends `AbstractMojo` and tells Maven via the `@Mojo` annotation what our goal name is. The body of the method takes care of whatever job we want. In this case, we simply log some text - but you have the full Java language and ecosystem available at this point to implement your goal.

To test the plugin in another project, we need to `mvn install` it, which will place our JAR into the local caching repository. Once there, we can pull our plugin into another project just like all

the other "real" plugins we've seen already in this chapter.

```
<build>
  <plugins>
    <plugin>
      <groupId>com.wellgrounded</groupId> ①
      <artifactId>wellgrounded-maven-plugin</artifactId> ①
      <version>1.0-SNAPSHOT</version>
      <executions>
        <execution> ②
          <phase>compile</phase>
          <goals>
            <goal>wellgrounded</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

- ① Reference our plugin coordinates
- ② Bind our goal to the `compile` phase

With this in place, we can see our plugin in action when we compile.

```
~: mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.wellgrounded:example >-----
[INFO] Building example 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] - maven-resources-plugin:2.6:resources (default-resources) -
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- wellgrounded-maven-plugin:1.0-SNAPSHOT:wellgrounded --- ①
[INFO] Extending Maven for fun and profit.
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  0.872 s
[INFO] Finished at: 2020-08-16T22:26:20+02:00
[INFO] -----
```

- ① Our plugin running as part of the `compile` phase

It's worth noting that if we simply include the plugin without the `<executions>` element we won't see our plugin show up anywhere in our project. Custom plugins must declare their desired phase in the lifecycle via the `pom.xml` file.

Visibility into the lifecycle and what goals are bound to what phases can be difficult, but the fortunately there's a plugin to help with that. `buildplan-maven-plugin` brings clarity to your current tasks.

Although it can be included in a `pom.xml` like any other plugin, a useful alternative to avoid repetition is putting it in your user's `~/.m2/settings.xml` file.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    https://maven.apache.org/xsd/settings-1.0.0.xsd">
  <pluginGroups>
    <pluginGroup>fr.jcgay.maven.plugins</pluginGroup>
  </pluginGroups>
</settings>
```

Once there, you can invoke it in any project building with Maven.

```
~: mvn buildplan:list

[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.wellgrounded:example >-----
[INFO] Building example 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] ---- buildplan-maven-plugin:1.3:list (default-cli) @ example ---
[INFO] Build Plan for example:

-----

| PLUGIN                | PHASE            | ID                  | GOAL        |
|-----------------------|------------------|---------------------|-------------|
| jacoco-maven-plugin   | initialize       | default             | prep-agent  |
| maven-compiler-plugin | compile          | default-compile     | compile     |
| maven-compiler-plugin | test-compile     | default-testCompile | testCompile |
| maven-surefire-plugin | test             | default-test        | test        |
| jacoco-maven-plugin   | test             | report              | report      |
| maven-jar-plugin      | package          | default-jar         | jar         |
| maven-failsafe-plugin | integration-test | default             | int-test    |
| maven-failsafe-plugin | verify           | default             | verify      |
| maven-install-plugin  | install          | default-install     | install     |
| maven-deploy-plugin   | deploy           | default-deploy      | deploy      |

-----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time:  0.461 s
[INFO] Finished at: 2020-08-30T15:54:30+02:00
[INFO] -----
```

NOTE If you don't want to add a plugin to your `pom.xml` or your `settings.xml`, you can just ask Maven to run a command using the fully qualified plugin name! In our example above we can just say `mvn fr.jcgay.maven.plugins:buildplan-maven-plugin:list` and Maven will download the plugin and run it one-off. This is great for uncommon tasks or experimentation.

Maven's [documentation for authoring plugins](#) is thorough and well maintained, so do take a look when starting to implement your own plugins.

Maven remains among the most common build tools for Java and has been hugely influential. However, not everyone loves its strongly opinionated stance. Gradle is the most popular alternative, so let's see how it tackles the same problem space.

11.3 Gradle

Gradle came onto the scene after Maven. That means it's compatible with much of the dependency management infrastructure Maven pioneered, but it takes a scripted approach to describing build *tasks* for those who feel overly constrained with Maven's XML configuration.

We'll take a closer look at that scripting in a little while, but let's get our feet wet by seeing how to run Gradle commands.

11.3.1 Installing Gradle

Gradle can be installed from its [website](#). Recent versions only rely on having JVM version 8 or greater. Once installed, you can run it at the command line and it will default to displaying help.

```
~: gradle
> Task :help

Welcome to Gradle 6.5.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>

For troubleshooting, visit https://help.gradle.org

BUILD SUCCESSFUL in 606ms
1 actionable task: 1 executed
```

This makes it easy to get started, but having a single global Gradle version isn't ideal. It is very common for a developer to build multiple different projects that could each different versions of Gradle.

To handle this, Gradle introduces the idea of a *wrapper*. The `gradle wrapper` task will capture a specific version of Gradle locally into your project. This is then accessed via the `./gradlew` or `gradlew.bat` commands. Many users of Gradle will rarely use the `gradle` command directly, and instead will work only with the `gradlew` wrapper.

NOTE

It's recommended that you include the `gradle` and `gradlew*` results of the wrapper in source control, but exclude the local caching of `.gradle`.

With the wrappers committed anyone downloading your project gets the properly versioned build tooling without any additional installs.

11.3.2 Tasks

Gradle's key concept is the *task*. A task defines a piece of work that can be invoked. Tasks can depend on other tasks, be configured via scripting, and added through Gradle's plugin system. These resemble Maven's goals, but are conceptually more open-ended.

Gradle provides excellent introspection features. Key among these is the `./gradlew tasks` meta-task which lists currently available tasks in your project. Before you've even declared anything, running `tasks` will present the following task list.

```
~: ./gradlew tasks
> Task :tasks

-----
Tasks runnable from root project
-----

Build Setup tasks
-----
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.

Help tasks
-----
buildEnvironment - Displays all buildscript dependencies in root project
components - Displays the components produced by root project.
dependencies - Displays all dependencies declared in root project.
dependencyInsight - Displays insight for dependency in root project
dependentComponents - Displays dependent components in root project
help - Displays a help message.
model - Displays the configuration model of root project. [incubating]
outgoingVariants - Displays the outgoing variants of root project.
projects - Displays the sub-projects of root project.
properties - Displays the properties of root project.
tasks - Displays the tasks runnable from root project.
```

Providing the `--dry-run` flag to any task will display the tasks Gradle would have run, without performing the actions. This is useful for understanding the flow of your build system, or debugging misbehaving plugins or custom tasks.

11.3.3 What's in a script?

The heart of a Gradle build is its *buildscript*. This is a key difference between Gradle and Maven - not only is the format different, but the entire philosophy is different too. Maven POM files are XML-based, whereas in Gradle the buildscript is an executable script written in a programming language - what's often referred to as a *domain specific language* or DSL. Modern versions of Gradle support both Groovy and Kotlin.

GROOVY VS KOTLIN

Gradle's DSL approach started out with Groovy. As we learned when we met it briefly in Chapter 8, Groovy is a dynamic language on the JVM and it fits nicely with the goal of flexibility and concise build scripting.

Since Gradle 5.0, however, another option has been available: Kotlin, which we covered in detail in Chapter 9.

NOTE

Kotlin build scripts use the extension `.gradle.kts` instead of `.gradle`.

This makes a lot of sense as Kotlin is now the dominant language for Android development - where Gradle is the platform's official build tool. Sharing the same language across all parts of your project can be a great simplifying factor.

For our purposes, Kotlin is also more like Java than Groovy. Narrowing this language gap means that if you're new to the Gradle ecosystem, it might make sense to write your build script with Kotlin if you are coding in Java.

Groovy remains a prominent and very viable option, but we're going to double down on our Kotlin experience and use it for all of our following examples. Anything we show in this chapter can be expressed similarly in a Groovy buildscript if you want to.

11.3.4 Using plugins

Gradle uses *plugins* to define everything about the tasks we use. As we saw above, listing tasks in a blank Gradle project doesn't say anything about building, testing, or deploying. All of that comes from plugins.

Numerous plugins ship with Gradle itself, so using them only requires a declaration in your `build.gradle.kts`. A key one is the `base` plugin.

```
plugins {
    base
}
```

A look at our tasks after including the `base` plugin reveals some common build lifecycle tasks that we might expect.

```

~:./gradlew tasks
> Task :tasks

-----
Tasks runnable from root project
-----

Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
clean - Deletes the build directory.

... Other tasks omitted for length

Verification tasks
-----
check - Runs all checks.

...

BUILD SUCCESSFUL in 640ms
1 actionable task: 1 executed

```

With that in place, let's get building a Gradle project for our code.

11.3.5 Building

While Gradle allows for customization to your heart's content, it defaults to expecting the same code layout that Maven established and popularized. For many (perhaps even most) projects it doesn't make sense to change this layout, although it is possible to do so.

Let's start with a basic Java library. To do this, we create the following source tree.

```

.
build.gradle.kts
gradle
    wrapper
        gradle-wrapper.jar
        gradle-wrapper.properties
gradlew
gradlew.bat
settings.gradle.kts
src
    main
        java
            com
                wellgrounded
                    AwesomeLib.java

```

- ① These files were automatically created by the `gradle wrapper` command

The `base` plugin doesn't know anything about Java, so we need a plugin with more awareness. For our use-case of a plain Java JAR, we'll use Gradle's `java-library` plugin. This plugin builds on all the necessary parts from `base` - in practice, you'll rarely see the `base` plugin alone in a Gradle build.

```
plugins {
    `java-library` ①
}
```

- ① Backticks (not apostrophes) are used around plugin names when they include special characters such as - here.

This yields a growing set of tasks in our build section.

```
Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and dependent projects.
buildNeeded - Assembles and tests this project and dependent projects.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.
```

In Gradle's terminology, `assemble` is the task which will compile and package up a JAR file. A dry run shows all of the steps, some of which the default tasks list doesn't show.

```
./gradlew assemble --dry-run
:compileJava SKIPPED
:processResources SKIPPED
:classes SKIPPED
:jar SKIPPED
:assemble SKIPPED
```

Running `./gradlew assemble` generates output in the build directory.

```
.
  build
    classes
      java
        main
          com
            wellgrounded
              Main.class
  libs
    wellgrounded.jar
```

MAKING AN APPLICATION

A plain JAR is a good start, but eventually you want to run an application. This takes more configuration, but again the pieces are available by default.

We'll change up our plugins and tell Gradle what the main class is for our application. We also can see several of the nice features that Kotlin brings in just this brief snippet.

```

plugins {
    java
    application
}

application {
    mainClassName = "wgjd.Main"
}

tasks.jar {
    manifest {
        attributes("Main-Class" to application.mainClassName)
    }
}

```

- ➊ Kotlin's optional parentheses when final argument is a lambda
- ➋ Plugin that knows how to compile Java
- ➌ Plugin that knows how to run a Java app
- ➍ Kotlin's clean property setting syntax
- ➎ Task for assembling JAR with a modified manifest
- ➏ Kotlin uses `to` syntax to declare a hash map in place (aka a *hash literal*)

Building with `./gradlew build` gets us the same JAR output as before, but if we execute `java -jar build/libs/wellgrounded.jar` our test program will run. Alternatively, the `application` plugin also supports `./gradlew run` to directly load and execute your main class for you.

NOTE

The `application` plugin only requires that `mainClassName` be set, but excluding the `tasks.jar` configuration will yield a JAR that `./gradlew run` knows how to start but `java -jar` doesn't. Definitely not recommended!

No person is an island. Similarly, few applications get far without pulling in other library dependencies. This is a major subject in Gradle and a point of considerable difference from Maven.

11.3.6 Dependencies in Gradle

To start introducing dependencies, we must first tell Gradle which repositories it can download from. There are built-in functions for `mavenCentral`, `jcenter` and `google`. You can use more detailed APIs to configure other repositories, including your own private instances.

```

repositories {
    mavenCentral()
}

```

We can then introduce our dependencies via the standard coordinate format popularized by Maven. Much like Maven had the `<scope>` element to control where a given dependency was

used, Gradle expresses this through *dependency configurations*. Each configuration tracks a particular set of dependencies. Your plugins define which configurations are available, and you add to a configuration's list with a function call. For example, to pull the [SLF4J library](#) to help with logging we'd use the following configurations.

```
dependencies {
    implementation("org.slf4j:slf4j-api:1.7.30")
    runtimeOnly("org.slf4j:slf4j-simple:1.7.30")
}
```

In this case, our code directly calls classes and methods in `slf4j-api`, so it is included via the `implementation` configuration. This makes it available during compilation and running the application. Our application should never directly call methods in `slf4j-simple`, though - that's done strictly through the `slf4j-api` - so requesting `slf4j-simple` as `runtimeOnly` ensures that code isn't available during compilation, preventing us from misusing the library.

Configurations can extend one another, much like deriving from a base class. Gradle applies this feature in many area, for example when creating classpaths. Gradle uses a `compileClasspath` and `runtimeClassPath` configuration. Those configurations in turn extend the `implementation` and `runtimeOnly` configurations to gather all the dependencies the user declares. From this list, Gradle can determine the proper classpath for a given situation.

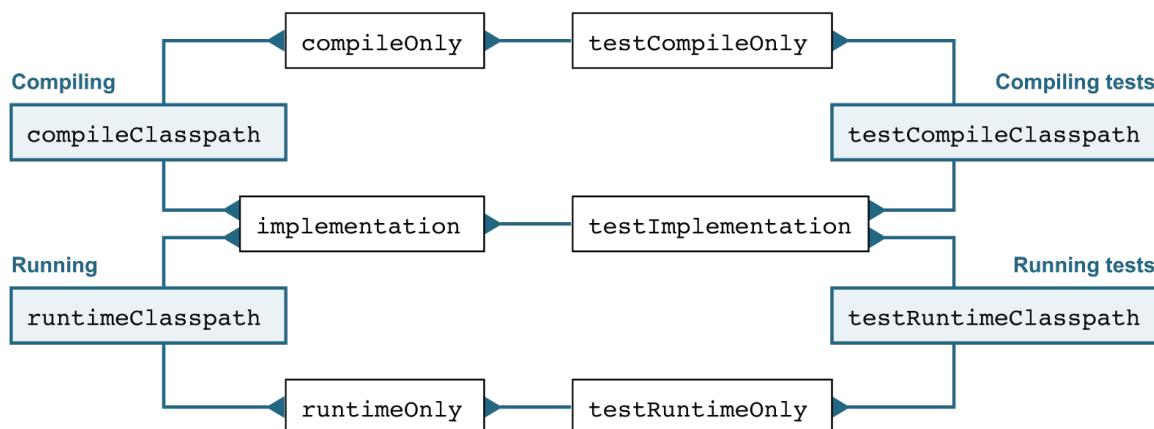


Figure 11.6 Hierarchy of Gradle configurations

Here are some primary configurations available when using the Java plugin that ships with Gradle, along with an indication of what other configurations each extends from.

Table 11.1 Typical Gradle Dependency Configurations

Name	Purpose	Extends
implementation	Primary dependencies used during compiling and running	
compileOnly	Dependencies only needed during compilation	
compileClasspath	Configuration Gradle uses to look up compilation classpath	compileOnly, implementation
compile	Deprecated configuration. Commonly replaced by using implementation.	
runtimeOnly	Dependencies only needed during runtime	
runtimeClasspath	Configuration Gradle uses to look up runtime classpath	runtimeOnly, implementation
runtime	Deprecated configuration. Commonly replaced by using runtimeOnly.	compile
testImplementation	Dependencies used during compiling and running tests	implementation
testCompileOnly	Dependencies only needed during compilation of tests	
testCompileClasspath	Configuration Gradle uses to look up test compilation classpath	testCompileOnly, testImplementation
testCompile	Deprecated configuration. Commonly replaced by using testImplementation.	compile
testRuntimeOnly	Dependencies only needed during runtime	runtimeOnly
testRuntimeClasspath	Configuration Gradle uses to look up test runtime classpath	testRuntimeOnly, testImplementation
testRuntime	Deprecated configuration. Commonly replaced by using testRuntimeOnly.	testCompile, runtime
archives	List of output JARs from our project	

Like Maven, Gradle uses the package information to create a transitive dependency tree. However, Gradle's default algorithm for handling version conflicts differs from Maven's "closest-to-the-root-wins" approach. When resolving, Gradle walks the full dependency tree to determine all the requested versions for any given package. From that the full set of requested versions, Gradle will then choose the *highest* available version.

This approach avoids some unexpected behavior in Maven's approach - for instance, changes in the ordering/depth of packages could result in different resolution. However, Gradle's technique doesn't provide as simple of an answer to the question of which version will be selected. It also opens the door to configurations where a valid resolution simply isn't available, and Gradle will fail to build rather than choosing a version which may cause problems.

Given this, Gradle provides rich APIs to override and control its resolution behavior. It also has

solid introspecting tools built-in to pull back the curtains when something goes wrong.

A key command when transitive dependency problems rear their ugly head is `./gradlew dependencies`.

```
~: ./gradlew dependencies

testImplementation - Implementation only dependencies for source set 'test'.
\--- junit:junit:4.12

testRuntimeClasspath - Runtime classpath of source set 'test'.
\--- junit:junit:4.12
    \--- org.hamcrest:hamcrest-core:1.3
```

In a large project this output can be overwhelming, so `dependencyInsight` lets you focus on the specific dependency you care about.

```
~: ./gradlew dependencyInsight \
    --configuration testRuntimeClasspath \
    --dependency hamcrest-core

> Task :dependencyInsight
org.hamcrest:hamcrest-core:2.2
variant "runtime" [
    org.gradle.status          = release (not requested)
    org.gradle.usage            = java-runtime
    org.gradle.libraryelements = jar
    org.gradle.category        = library

    Requested attributes not found in the selected variant:
        org.gradle.dependency.bundling = external
        org.gradle.jvm.version       = 11
]
Selection reasons:
- By conflict resolution : between versions 2.2 and 1.3

org.hamcrest:hamcrest-core:2.2
\--- testRuntimeClasspath

org.hamcrest:hamcrest-core:1.3 -> 2.2
\--- junit:junit:4.12
    \--- testRuntimeClasspath
```

Dependency conflicts can be hard to resolve. The best approach if possible is to use the dependency tools in Gradle to find mismatches and upgrade to mutually compatible versions. Ah, to live in a world where that was always possible!

In some cases, you may simply need Gradle to ignore a higher version it found requested in the tree. This can be accomplished in a couple of ways. The first is to exclude the group

```
dependencies {
    testImplementation("junit:junit:4.12")                               ①
    testImplementation("com.wellgrounded:newer-hamcrest-helper:1.0") {
        exclude(group = "org.hamcrest")                                     ②
    }
}
```

- ① Dependency from `junit` will be chosen
- ② We tell Gradle to ignore the `org.hamcrest` dependencies from our helper

This requires determining the specific dependencies which bring unacceptable versions to the mix, though. You can instead tell Gradle more generally a *constraint* about versions you want it to add to its resolution mix.

```
dependencies {
    testImplementation("junit:junit:4.12")          ①
    testImplementation("com.wellgrounded:newer-hamcrest-helper:1.0") ①
    constraints {
        testImplementation("org.hamcrest:hamcrest-core:1.3") {
            because("A newer version isn't compatible because...") ②
        }
    }
}
```

- ① All dependencies just ask for what they want as before
- ② Gradle will respect this constraint or fail the resolution
- ③ It's good practice to use `because` for documenting why we're intervening because Gradle's tooling can use that text, versus comments in the script which are only useful to human readers.

If you really need to get precise, you can set a strict version which will override any other resolution.

```
dependencies {
    testImplementation("junit:junit:4.12")          ①
    testImplementation("com.wellgrounded:newer-hamcrest-helper:1.0") ①
    testImplementation("org.hamcrest:hamcrest-core") {
        version {
            strictly("1.3")
        }
    }
}
```

- ① All dependencies just ask for what they want as before
- ② Force version 1.3. This won't match 1.3.1 or any other related version.

As we've mentioned in prior sections, manually forcing dependency versions is a last resort and deserves special attention to ensure you aren't getting runtime exceptions. A robust test suite can be critical to save time when ensuring your mix of libraries works together smoothly.

11.3.7 Adding Kotlin

As we've discussed in both Chapter 8 and the Maven section of this chapter, the ability to add another language to a project is a huge benefit of running on the JVM.

Adding Kotlin shows off the benefits that Gradle's scripted approach over Maven's more static XML-based configuration. Following the standard multi-lingual layout from our original code yields this.

```
.
build.gradle.kts
gradle
    wrapper
        gradle-wrapper.jar
        gradle-wrapper.properties
gradlew
gradlew.bat
settings.gradle.kts
src
    main
        java
            com
                wellgrounded
                    Main.java
    kotlin
        com
            wellgrounded
                kotlin
                    MessageFromKotlin.kt
test
    java
        com
            wellgrounded
                MainTest.java
```

①

- ① Our additional Kotlin code appears under the `kotlin` subdirectories.

We enable Kotlin support via a Gradle plugin in our `build.gradle.kts`.

```
plugins {
    application
    java
    id("org.jetbrains.kotlin.jvm") version "1.4.31"
}
```

And that's it. Because of Gradle's flexibility, the plugin is able to alter the build order and add the necessary `kotlin-stdlib` dependencies without us having to take additional steps.

11.3.8 Testing

The `assemble` task we first discussed will compile and package your main code but we need to compile and run our tests as well. The `build` task is configured by default for just that.

```
./gradlew build --dry-run
:compileJava SKIPPED
:processResources SKIPPED
:classes SKIPPED
:jar SKIPPED
:assemble SKIPPED
:compileTestJava SKIPPED
:processTestResources SKIPPED
:testClasses SKIPPED
:test SKIPPED
:check SKIPPED
:build SKIPPED
```

We'll add a test case using the standard locations.

```
src
test
java
com
    wellgrounded
        MainTest.java
```

Next up, we're need to add our test framework to the right dependency configuration to make it available to our code.

```
dependencies {
    ...
    testImplementation("junit:junit:4.12")
}
```

The `testImplementation` configuration makes `junit` available when building and executing test - but not main - code. When we run the `./gradlew build` next, you'll see that it's downloading the library into our local cache if it wasn't already there.

Test output for JUnit can be a little sparse, but full listings with stack traces are generated under `build/reports/test`. In general, if you aren't seeing output on the console that you want when building, it's buried somewhere under `build`.

11.3.9 Review and analysis

The build is a great place to add functionality to protect your project. One type of check beyond unit testing is static analysis. There are several tools in this category, but [SpotBugs](#) (the successor to FindBugs) is an easy one to get started with.

```
plugins {
    java
    application
    id("com.github.spotbugs") version "4.3.0"
}
```

If we deliberately introduce a problem in our code (for instance, implementing `equals` on a class without also overriding `hashCode`), then a typical `./gradlew check` will let us know there's a problem.

```

~:./gradlew check

> Task :spotbugsTest FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':spotbugsTest'.
> A failure occurred while executing SpotBugsRunnerForWorker
  > Verification failed: SpotBugs violation found:
    2. SpotBugs report can be found in build/reports/spotbugs/test.xml

* Try:
Run with --stacktrace option to get the stack trace.
Run with --info or --debug option to get more log output.
Run with --scan to get full insights.

* Get more help at https://help.gradle.org

BUILD FAILED in 1s
5 actionable tasks: 3 executed, 2 up-to-date

```

As with unit testing failures, report files are under `build/reports/spotbugs`. Out of the box it may only generate an XML file which, while nice for computers, is less useful to most people. We can configure the plugin to emit HTML for us.

```

tasks.withType<com.github.spotbugs.snom.SpotBugsTask>()      ①
    .configureEach {                                         ②
        reports.create("html") {
            isEnabled = true
            setStylesheet("fancy-hist.xsl")
        }
    }

```

- ① `tasks.withType` looks up tasks for us in a type safe manner
- ② `configureEach` runs the block as if we had written `tasks.spotbugsMain { }` then `tasks.spotbugsTest { }` with the same code.
- ③ Remaining configuration is taken from the project's [README on GitHub](#)

11.3.10 Moving beyond Java 8

In Chapter 1, we noted a series of modules that belonged with Java Enterprise Edition, but were present in the core JDK. These were deprecated with JDK 9 and removed with JDK 11 but remain available as external libraries.

- `java.activation` (JAF)
- `java.corba` (CORBA)
- `java.transaction` (JTA)
- `java.xml.bind` (JAXB)
- `java.xml.ws` (JAX-WS, plus some related technologies)
- `java.xml.ws.annotation` (Common Annotations)

If your project relies on any of these modules, your build might break when you move to a

more recent JDK. Fortunately there are a few simple dependency additions in your `build.gradle.kts` to address the issue.

```
dependencies {
    implementation("com.sun.activation:jakarta.activation:1.2.2")
    implementation("org.glassfish.corba:glassfish-corba-omgapi:4.2.1")
    implementation("javax.transaction:javax.transaction-api:1.3")
    implementation("jakarta.xml.bind:jakarta.xml.bind-api:2.3.3")
    implementation("jakarta.xml.ws:jakarta.xml.ws-api:2.3.3")
    implementation("jakarta.annotation:jakarta.annotation-api:1.3.5")
}
```

11.3.11 Using Gradle with modules

Like Maven, Gradle supports the JDK module system fully. Let's break down what we need to alter to use our modular projects with Gradle.

A MODULAR LIBRARY

A modular library typically has two major structural differences - the change from using `main` to the module name in the directory under `src`, and the addition of a `module-info.java` file at the root of our module.

```
.
  build.gradle.kts
  gradle
    wrapper
      gradle-wrapper.jar
      gradle-wrapper.properties
  gradlew
  gradlew.bat
  settings.gradle.kts
  src
    com.wellgrounded.modlib           ①
      java
        com
          wellgrounded
            hidden               ②
              CantTouchThis.java
            visible              ③
              UseThis.java
    module-info.java                  ④
```

- ① Directory name aligned with our module
- ② We intend to keep this package hidden
- ③ This package will be exported for use outside this module
- ④ The `module-info.java` declarations for this module

Gradle doesn't automatically find our altered source location, so we need to give it a hint in `build.gradle.kts` where to look.

```
sourceSets {
    main {
        java {
            setSrcDirs(listOf("src/com/wellgrounded/modlib/java"))
        }
    }
}
```

The `module-info.java` file contains the typical declarations that we saw demonstrated earlier in this chapter and in Chapter 2. We'll name our module and select one, but not both, of our packages to export.

```
module com.wellgrounded.modlib {
    exports com.wellgrounded.modlib.visible;
}
```

That's all that's required to make our library consumable as a module. Next we'll use the library from a modular app.

A MODULAR APPLICATION

When we set out to test our modular application under Maven, the simplest way to share the library we'd created with our app was to install it to the local Maven repository. This is also easily supported from Gradle via the `maven-publish` plugin, but we also have another option.

Our modular application has a standard layout as shown below. For ease of testing, we'll make sure the top level directories live next to each other.

```
mod-lib
...
mod-app
build.gradle.kts
gradle
wrapper
    gradle-wrapper.jar
    gradle-wrapper.properties
gradlew
gradlew.bat
settings.gradle.kts
src
    com.wellgrounded.modapp
        java
            com
                wellgrounded
                    Main.java
    module-info.java
```

- ① mod-lib library source at the same level as our mod-app application.
- ② Directory name aligned with module name
- ③ `module-info.java` to declare this a modularized application

Our `module-info.java` tells our name and module requirements.

```
module com.wellgrounded.modapp {           ①
    requires com.wellgrounded.modlib;      ②
}
```

- ① Our module name
- ② Our requirement on our library's exported packages

For testing our local library, rather than installing it, for the moment we'll refer to it locally. This can be accomplished using the `files` function in the spot where we'd previously have given the GAV coordinates for our dependency. This obviously won't work once we're ready to start sharing and deploying, but it's a quick move to get our local testing started.

```
dependencies {
    implementation(files("../mod-lib/build/libs/gradle-mod-lib.jar"))
}
```

Next up, current versions of Gradle require a hint we want it to sniff out which dependencies are modular to properly put them on the module path instead of the classpath. This may become a default eventually, but at the time of this writing (Gradle 6.8.x) it remains an opt-in.

```
java {
    modularity.inferModulePath.set(true)
}
```

Last and most mundanely, like our library we need to let Gradle know about our non-Maven standard file location.

```
sourceSets {
    main {
        java {
            setSrcDirs(listOf("src/com.wellgrounded.modapp/java"))
        }
    }
}
```

With all this in place, `./gradlew build run` has the expected result. If we attempt to use a package from the library which isn't exported, we confront the error at compilation time.

```
> Task :compileJava FAILED
/mod-app/src/com.wellgrounded.modapp/java/com/wellgrounded/Main.java:4:
error: package com.wellgrounded.modlib.hidden is not visible

import com.wellgrounded.modlib.hidden.CantTouchThis;
^
(package com.wellgrounded.modlib.hidden is declared in module
com.wellgrounded.modlib, which does not export it)
1 error
```

JLINK

A capability we saw in Chapter 2 that modules unlock is the ability to create a streamlined environment for an application to work in, with only the dependencies it requires. This is possible because the module system gives us concrete guarantees about which modules our code uses, so tooling can construct the necessary, minimal set of modules.

NOTE

JLink can only work with fully modularized applications. If an application is still loading some code via the classpath, JLink can't succeed in making a safe, complete image.

This feature is most evident through the `jlink` tool. For a modular application, JLink can produce a fully functioning JVM image that can be run without depending on a system-installed JVM.

Let's revisit the application from Chapter 2 that we demonstrated JLink with to see how Gradle plugins streamline the management. The sample application, available in the supplement, uses JDK classes to attach to all the running JVM processes on a machine and display various information about them.

As a modular application we're going to package, an important bit to review is the application's own `module-info.java` declarations. These tell us what JLink will need to pull into its custom image for our build to work.

```
module wgjd.discovery {
    exports wgjd.discovery;

    requires java.instrument;
    requires java.logging;
    requires jdk.attach;
    requires jdk.internal.jvmstat; ①
}
```

- ① Red flag, note the `jdk.internal` package that we're reaching into!

Before we even get started with JLink, moving from hand-compiling to our Gradle build takes a little extra configuration. We need to apply the same modularization changes explained in the preceding section as a start. But even once those are in place, we can't compile successfully.

```

~:./gradlew build

> Task :compileJava FAILED
/gradle-jlink/src/wgjd.discovery/wgjd/discovery/VMIntrospector.java:4:
error: package sun.jvmstat.monitor is not visible
import sun.jvmstat.monitor.MonitorException;
^
(package sun.jvmstat.monitor is declared in module jdk.internal.jvmstat,
 which does not export it to module wgjd.discovery)

/gradle-jlink/src/wgjd.discovery/wgjd/discovery/VMIntrospector.java:5:
error: package sun.jvmstat.monitor is not visible
import sun.jvmstat.monitor.MonitoredHost;
^
(package sun.jvmstat.monitor is declared in module jdk.internal.jvmstat,
 which does not export it to module wgjd.discovery)

/gradle-jlink/src/wgjd.discovery/wgjd/discovery/VMIntrospector.java:6:
error: package sun.jvmstat.monitor is not visible
import sun.jvmstat.monitor.MonitoredVmUtil;
^
(package sun.jvmstat.monitor is declared in module jdk.internal.jvmstat,
 which does not export it to module wgjd.discovery)

/gradle-jlink/src/wgjd.discovery/wgjd/discovery/VMIntrospector.java:7:
error: package sun.jvmstat.monitor is not visible
import sun.jvmstat.monitor.VmIdentifier;
^
(package sun.jvmstat.monitor is declared in module jdk.internal.jvmstat,
 which does not export it to module wgjd.discovery)

4 errors

FAILURE: Build failed with an exception.

```

The module system is letting us know that we're breaking the rules by trying to use classes that are in `jdk.internal.jvmstat`. Our module, `wgjd.discovery` is not included in the `jdk.internal.jvmstat` list of allowed modules. Understanding the rules and the risks we're taking, we can use `--add-exports` though to force our module into the list. This is done via a compiler flag, and looks like this in our Gradle configuration:

```

tasks.withType<JavaCompile> {
    options.compilerArgs = listOf(
        "--add-exports",
        "jdk.internal.jvmstat/sun.jvmstat.monitor=wgjd.discovery"
    )
}

```

With that we get a clean compile and we can turn to using JLink to package it up. The plugin with the most mindshare today is `org.beryx.jlink`, known in the documentation as "[The Badass JLink Plugin](#)". We add it to our Gradle project with a plugin line.

```

plugins {
    id("org.beryx.jlink") version("2.23.3") ①
}

```

- ① This plugin automatically applies `application` for us, so we don't need to repeat that declaration.

After adding that, we'll see a `jlink` task in our list which we can run straight away. The result will show up in the `build/image` directory.

```
build/image/
bin
  gradle-jlink
  gradle-jlink.bat
  java
  keytool
conf
  ... various configuration files
include
  ... require headers
legal
  ... license and legal information for all included modules
lib
  ... library files and dependencies for our image
release
```

The `build/image/bin/java` is our custom JVM with only our application's module dependencies available to it. You can run it just like you would your normal `java` command from the terminal.

```
~:build/image/bin/java -version
openjdk version "11.0.6" 2020-01-14
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.6+10)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.6+10, mixed mode)
```

We can pass `build/image/bin/java` our module to start up, but the plugin has neatly generated a startup script at `build/image/bin/gradle-jlink` (named after our project) that we can use instead. But not all is well with our newly minted image.

```
~:build/image/bin/gradle-jlink

Java processes:
PID  Display Name      VM Version      Attachable
Exception in thread "main" java.lang.IllegalAccessError:
  class wgjd.discovery.VMIntrospector (in module wgjd.discovery) cannot
  access class sun.jvmstat.monitor.MonitorException (in module
  jdk.internal.jvmstat) because module jdk.internal.jvmstat does not
  export sun.jvmstat.monitor to module wgjd.discovery
wgjd.discovery/wgjd.discovery.VMIntrospector.accept(VMIntrospector.java:19)
wgjd.discovery/wgjd.discovery.Discovery.main(Discovery.java:26)
```

This isn't an entirely unfamiliar error message - it's another flavor of the same access issue we solved with the compiler options before. Apparently we need to inform the application startup of our module cheating needs as well. Fortunately, the plugin has extensive configuration for the parameters both to run `jlink` and for the resulting scripts created for us.

```
jlink {
    launcher{
        jvmArgs = listOf(
            "--add-exports",
            "jdk.internal.jvmstat/sun.jvmstat.monitor=wgjd.discovery"
    }
}
```

With that addition, the startup script gets everything running.

```
~:build/image/bin/gradle-jlink
Java processes:
PID  Display Name      VM Version   Attachable
833  wgjd.discovery/wgjd.discovery.Discovery  11.0.6+10    true
276  org.jetbrains.jps.cmdline.Launcher /Applications/IntelliJ IDEA CE.app...
```

It's worth noting that the image we generated here defaults to targeting the same operating system as JLink is running on. However, that isn't required - cross-platform support is available. The primary requirement is that you have the files from the target platform's JDK installation available. These are easily available from sources such as the [AdoptOpenJDK website](#).

```
jlink {
    targetPlatform("local", System.getProperty("java.home"))           ①
    targetPlatform('linux-x64', '/linux_jdk-11.0.10+9')                ②

    launcher{
        jvmArgs = listOf(
            "--add-exports",
            "jdk.internal.jvmstat/sun.jvmstat.monitor=wgjd.discovery")
    }
}
```

- ① Build an image based on whatever the local JDK is
- ② Build an image pointed to a Linux JDK we've downloaded

Once you start targeting specific platforms, the plugin will put additional directories in the build/image results. Obviously, you'll have to take those results to a matching system to test them.

A final roadblock that may come up in trying to use JLink is its restrictions around automatically named modules. While the feature to just add a name into the JAR manifest and get some basic ability to participate in the modular world is great for migrations, JLink sadly doesn't support it.

The Badass JLink Plugin, though, has you covered. It will repackage any automatically named modules into a proper module which JLink can consume. [The documentation gives full coverage to this feature](#) which may be the difference between JLink working or not depending on your application's dependencies.

11.3.12 Customizing

One of Gradle's biggest strength is its open-ended flexibility. Without pulling in plugins, it doesn't even have a concept of a build lifecycle. You can add tasks and reconfigure existing tasks with few restrictions. There's no need to keep a scripts directory around in your project with random tooling - your custom needs can be integrated right into your day-to-day to build and testing tool.

CUSTOM TASKS

Defining a custom task can be done directly in your `build.gradle.kts` file.

```
tasks.register("wellgrounded") {
    println("configuring")
    doLast {
        println("Hello from Gradle")
    }
}
```

Running this will produce the following output.

```
~: ./gradlew wellgrounded
configuring...
> Task :wellgrounded
Hello from Gradle
```

The `println("configuring")` line is run during setup of the task, but the contents of the `doLast` block happens when the task actually ran. We can confirm this by doing a dry-run on our task.

```
~: ./gradlew wellgrounded --dry-run
configuring...
:wellgrounded SKIPPED
```

Tasks can be configured to depend on other tasks.

```
tasks.register("wellgrounded") {
    println("configuring...")
    dependsOn("assemble")
    doLast {
        println("Hello from Gradle")
    }
}
```

This technique applies equally well to tasks you didn't author - you can look them up and add your task as a dependency.

```
tasks {
    named<Task>("help") {
        dependsOn("wellgrounded")
    }
}
```

```

~: ./gradlew help
configuring...

> Task :wellgrounded
Hello from Gradle

> Task :help

Welcome to Gradle 6.5.

To run a build, run gradlew <task> ...

To see a list of available tasks, run gradlew tasks

To see a list of command-line options, run gradlew --help

To see more detail about a task, run gradlew help --task <task>

For troubleshooting, visit https://help.gradle.org

```

CREATING CUSTOM PLUGINS

Custom tasks in your build file are a great start, but maybe you want to test that code or share it between projects. Gradle plugins are built for just that.

Gradle plugins are implemented as JVM code. They can be provided directly in your project as source files, or they can be pulled in through libraries. Many plugins have been written in Groovy, as the original scripting language supported by Gradle, but you can do it in any JVM language.

Plugins can be coded directly in your buildscript, and we'll demonstrate the main APIs using that technique. When you're ready to share, you can pull the code into its own separate project.

Here is an equivalent to our earlier `wellgrounded` task.

```

class WellgroundedPlugin : Plugin<Project> {      ①
    override fun apply(project: Project) {
        project.task("wellgrounded") {                  ②
            doLast {
                println("Hello from Gradle")
            }
        }
    }
}

apply<WellgroundedPlugin>()                         ③

```

- ① Derives from `Plugin`
- ② Uses familiar project level API and task implementation
- ③ `apply` to actually use the plugin—it isn't automatically invoked like our earlier task definition.

Apart from sharing, a strength of authoring tasks as plugins is the ability to provide more configuration. Gradle does this through `Extension` objects.

```

open class WellgroundedExtensions {
    var count: Int = 1
}

class WellgroundedPlugin : Plugin<Project> {
    override fun apply(proj: Project) {
        val extensions = proj.extensions
        val ext = extensions.create<WellgroundedExtensions>("wellgrounded")
        proj.task("wellgrounded") {
            doLast {
                repeat(ext.count) {
                    println("Hello from Gradle")
                }
            }
        }
    }
}

apply<WellgroundedPlugin>()

configure<WellgroundedExtensions> {
    count = 4
}

```

All the power of our programming language is available within our plugins.

If you extract a plugin to another library, you can include it in your build through the same mechanism we saw earlier for including the spotbugs plugin.

```

plugins {
    id("com.wellgrounded.gradle") version "1000.0"
}

apply<WellgroundedPlugin>()

configure<WellgroundedExtensions> {
    count = 4
}

```

11.4 Summary

- Build tools are central to how Java software gets constructed in the real world. They automate tedious operations, help with dependency management, and ensure that developers are doing their work consistently.
- Maven and Gradle are the two most common build tools in the Java ecosystem, and most tasks can be accomplished in either.
- Maven takes an approach of configuration via XML combined with plugins written in JVM code.
- Gradle provides a scripted approach (in Kotlin or Groovy) for defining buildscripts, while also supporting plugins in full JVM code
- Dealing with conflicting dependencies is a major topic whatever your build tool, and both Maven and Gradle give you a variety of ways to work out when your library versions disagree.
- Modules as seen in Chapter 2 require some changes to our build scripting and source code layout, but these are well-supported by the tooling.

Advanced functional programming

14

This chapter covers

- Functional Programming Concepts
- Limits of FP in Java
- Kotlin advanced FP
- Clojure advanced FP

We have already met functional programming concepts earlier in the book, but in this chapter we want to draw together the threads and step it up.

There is a lot of talking about *functional programming* in the industry but it remains a rather ill-defined concept.

The sole point that is agreed upon is that in a functional programming language (FP) code is representable as a first class data item - i.e. that it should be possible to represent a piece of deferred computation as a value that can be assigned to a variable.

This definition is, of course, ludicrously broad - for all practical purposes every mainstream language (with very few exceptions) in the last 30 years meets this definition. So, when different groups of programmers discuss FP, they are talking about different things. Each tribe has a different, tacit understanding of what other language properties are implicitly understood to *also* be included under the term "FP".

In other words - just as with OO - there is no fundamentally agreed-upon definition of what a "functional programming language" is. Alternatively - if everything is a FP language, then nothing is.

The well-grounded developer is well advised to visualize programming languages upon an axis

(or better yet, as a point in a multi-dimensional space of possible language characteristics). Languages are simply more or less functional than other languages - there is not some absolute scale that they are weighed against.

Let's meet some of the concepts of the common toolbox of functional programming languages that go beyond the somewhat facile "code is data" notion.

14.1 Introduction to functional programming concepts

In what follows we will frequently speak of *functions* - but neither the Java language nor the JVM has any such thing - all executable code must be expressed as a *method*, which is defined, linked and loaded within a *class*.

Other, non-JVM languages, however, have a different conception of executable code - so when we refer to a function in this chapter, it should be understood that we mean a piece of executable code that roughly corresponds to a Java method.

14.1.1 Pure functions

A *pure function* is a function that does not alter state of any other entity. It is sometimes said to be *side-effect free* - which is intended to mean that the function behaves like an ideation of a mathematical function - it takes in arguments, does not affect them in any way, and returns a result that is dependent only on the values that have been passed.

Related to the concept of purity is the idea of *referential transparency*. This is somewhat unfortunately named - it has nothing to do with references as a Java programmer would understand them. Instead, it means that a function call can be replaced with the result of any previous call to the same function with the same arguments.

It is obvious that all pure functions are referentially transparent, but there may also exist functions that are not pure and yet are also referentially transparent. To allow a non-pure function to be considered in this way would require a formal proof based on code analysis.

Purity is about code - but immutability is about data, and that's the next FP concept we will look at.

14.1.2 Immutability

Immutability means that after an object has been created, its state cannot be altered.

The default in Java is for objects to be mutable. The keyword `final` is used in various ways in Java - but the one that concerns us here is to prevent modification of fields after creation. Other languages may favour immutability and indicate that preference in various ways - such as Rust, which requires programmers to explicitly make variables mutable with the `mut` modifier.

Immutability makes code easier to reason about - objects have a trivial state model, simply because they are constructed in the only state they will ever exist in. Among other benefits, this means that they can be copied and shared safely - even between threads.

NOTE

We might ask whether there are any "almost immutable" approaches to data that still maintain some (or most) of the attractive properties of immutability. In fact, the Java `CompletableFuture` class that we have already met is one such example. We'll have more to say in the next chapter.

One consequence is that, as immutable objects cannot be altered then the only way that state change can be expressed in a system is by starting from an immutable value and constructing a completely new immutable value that is more-or-less the same, but with some fields altered - possibly by the use of *withers* (aka `with*` methods).

For example, the `java.time` API makes very extensive use of immutable data, and new instances can be created by the use of withers:

```
LocalDate ld = LocalDate.of(1984, Month.APRIL, 13);
LocalDate dec = ld.withMonth(12);
System.out.println(dec);
```

The immutable approach has consequences - specifically a potentially large impact on the memory subsystem, as the components of the old value have to be copied as part of the creation of the modified value. This means that in-place mutation is often much cheaper from a performance perspective.

14.1.3 Higher-order functions

Higher-order functions are actually a very simple concept, which follows from the following insight: If a function can be represented as a data item, then it should be able to be treated as though it was any other value.

We can define a *higher-order function* as a function value that does one or more of:

- Taking a function value as a parameter
- Returning a function value

Consider, for example, a static method that takes in a Java `String` and generates a function object from it:

```
public static Function<String, String> makePrefixer(String prefix) {
    return s -> prefix +": "+ s;
}
```

This provides a straightforward way to make function objects. Let's now combine it with another

static method, that this time accepts a function object as input:

```
public static String doubleApplier(String input,
                                    Function<String, String> f) {
    return f.apply(f.apply(input));
}
```

This provides us with a simple example:

```
var f = makePrefixer("NaNa");
System.out.println(doubleApplier("Batman", f));
```

- ① Create a function object
- ② Pass the function object as a parameter to another method

However, this is not quite the whole story for Java - as we will see in the next section.

14.1.4 Recursion

A *recursive* function is one that calls itself on at least some of the code paths through the function.

This leads to one of the oldest jokes in programming: "In order to understand recursion, one must first understand recursion."

However, to be more strictly accurate, we might write it as:

In order to understand recursion, one must first understand

1. recursion and
2. that in a physically realizable system, every chain of recursive calls must eventually terminate and return a value.

The second point is important - programming languages use call stacks to allow functions to call other functions, and this occupies space in memory. Recursion therefore has the problem that deep recursive calls may potentially use up too much memory and crash.

In terms of theoretical computer science, recursion is interesting and important for many different reasons. One of the most important is that recursion can be used as a basis to explore theories of computation, and ideas such as *Turing completeness*, which is loosely the idea that all non-trivial computation systems have the same theoretical capability to perform calculations.

14.1.5 Closures

A *closure* is usually defined as a lambda expression that "captures" some state from the surrounding context. However, for this definition to make sense we need to explain the meaning of the capturing concept.

When we create a value and assign (or bind) it to a local variable, then the variable will exist and can be used until some later point in the code. This later point may well be the end of the function or block where the variable was declared. The area of code where the variable exists and can be used is the *scope* of the variable.

When we create a function value, then the local variables declared within the function body will still be in scope during the invocation of the function value, which will occur later than the point where the function value is declared.

If, in the declaration of the function value, we mention a variable (or other state, such as a field) that is declared outside the scope of the function body, then the function value is said to have *closed over* the state, and the function value is known as a closure.

When the closure is later invoked, then it has full access to the captured variables - even if the invocation happens in a different scope to that in which the capture was declared.

For example, in Java:

```
public static Function<Integer, Integer> closure() {
    var atomic = new AtomicInteger(0);
    return i -> atomic.addAndGet(i);
}
```

This static method is a higher-order function that returns a Java closure - because it returns a lambda expression that references `atomic` - which was declared as a local variable inside the method, i.e. in the scope where the lambda was itself declared.

The closure returned from `closure()` can be called repeatedly, and it will aggregate state on each call.

14.1.6 Laziness

We briefly mentioned the concept of *laziness* in Chapter 10. Essentially, *lazy evaluation* allows the computation of the value of an expression to be deferred until the value is actually required. By contrast, the immediate evaluation of an expression is known as *eager evaluation* (or *strict evaluation*).

The idea of laziness is simple - if you don't need to do work, don't do it! It sounds simple, but has deep ramifications for how you write your programs and how they perform. A key part of this additional complexity is that your program needs to track what work has and hasn't been completed already.

Not every language supports lazy evaluation, and many programmers may have only encountered eager evaluation at this point in their journey - and that's completely OK.

For example, there is no general language-level support for laziness in Java, so it's difficult to give a clear example of the feature. We'll have to wait until we talk about Kotlin to make it concrete.

However, whilst laziness is not necessarily a natural concept for a Java programmer, laziness is an extremely useful and powerful technique in FP. In fact there are FP languages, such as Haskell, where lazy evaluation is the default.

14.1.7 Currying and partial application

Currying, unfortunately, has nothing to do with food. Instead, it is a programming technique named after Haskell Curry (who also gave his name to the Haskell programming language). To explain it, let's start with a concrete example.

Consider an eagerly-evaluated, pure function that takes two arguments. If we supply both arguments, then we will get a value, and the function call can be replaced everywhere by the resulting value (this is referential transparency). But what happens if we supply not both, but only one of the two arguments?

Intuitively, we can think of this as creating a new function - but one that only needs a single argument to calculate a result. This new function is called a *curried function* (or *partially-applied function*).

Java does not have direct support for currying, so we will again defer making a concrete example until later in the chapter.

Looking further afield, some programming languages support the notion of functions with multiple argument lists (or have syntax that allows the programmer to fake them). In this case, then another way to think about currying is as a transformation of functions.

In mathematical notation, we are translating a multiple-argument function that is called as $f(a, b)$ into one callable as $(g(a))(b)$, where $g(a)$ is the partially-applied function.

As should be obvious now, the different languages we've met so far have different levels of support for functional programming - for example, Clojure has very good support for many of the concepts that we have discussed in this section.

Java, on the other hand, is a very different story, as we'll see in the next section.

14.2 Limitations of Java as a FP language

Let's start with the good news, such as it is.

Java definitely clears the rather low bar of "represent code as data", via the types in

`java.util.function` and also via the extensive introspective support the runtime provides (such as Reflection and Method Handles).

NOTE The use of inner classes to simulate function objects as a technique predates Java 8 and was present in libraries like Google's Guava, so strictly speaking, Java's ability to represent code as data is not tied to that version.

Since version 8, the Java language goes somewhat further than the bare minimum - with the introduction of streams and with them a heavily restricted domain of lazy operations.

However, despite the arrival of streams, Java is not a naturally functional environment. Some of this is due to the history of the platform and what are - by now - decades old design decisions.

NOTE It is worth remembering that Java is a 25-year old imperative language that has been iterated upon extensively. Some of its APIs are amenable to FP, immutable data, etc - and some are not. This is the reality of working in a language that has survived, and thrived, yet still remains backwards compatible.

So, overall, Java is perhaps best described as a "slightly functional programming language". It has the basic features needed to support FP, and provides developers with access to basic patterns like filter-map-reduce via the Streams API - but most advanced functional features are either incomplete or missing entirely. Let's take a detailed look.

14.2.1 Pure functions

As we saw in Chapter 4, Java's bytecodes do many different sorts of things - including arithmetic, stack manipulation, flow control and especially invocation and data storage and retrieval.

For well-grounded developers who already understand JVM bytecode this means that we can express purity of methods by thinking about the effect of bytecodes.

Specifically, a pure method in a JVM language is one that:

- Does not modify object or static state (does not contain `putfield` or `putstatic`)
- Does not depend upon external mutable object or static state
- Does not call any non-pure method

This is a pretty restrictive set of conditions, and underlines the difficulty of using the JVM as a basis for pure functional programming.

There is also a question about the semantics - i.e. the intent - of the different interfaces present in

the JDK. For example, `Callable` (in `java.util.concurrent`) and `Supplier` (in `java.util.function`) both do basically the same thing - they perform some calculation and return a value.

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}

@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

They are both `@FunctionalInterface` and are both routinely used as the target type for a lambda. The signatures of the interfaces are the same, apart from different approaches to handling exceptions.

However, they can be seen as having different roles - a `Callable` implies potentially non-trivial amounts of work in the called code to create the value that will be returned. On the other hand, the name `Supplier` seems to imply less work - perhaps just returning a cached value.

14.2.2 Mutability

Java is a mutable language - it is baked into its design from the earliest days.

Partly this is an accident of history - the machines of the late 90s (from when Java hails) were very restricted (by modern standards) in terms of memory. An immutable data model would have greatly increased stress on the memory management subsystem, and caused much more frequent GC events - leading to far worse throughput.

Java's design therefore favours mutation over creation of modified copies. So in-place mutation can be seen as a design choice caused by performance tradeoffs from 25 years ago.

The situation is, however, even worse than that. Java refers to all composite data by reference, and the `final` keyword applies to the reference, *not* to the data. E.g. when applied to fields, all this means is that the field can only be assigned to once.

This means that even if a object has all final fields, the composite state can still be mutable - because the object can hold a `final` reference to another object that has some non-final fields. This leads to the problem of shallow immutability, as we discussed in Chapter 5.

NOTE	For the C++ programmers - this essentially means that Java has no concept of <code>const</code> - although it does have it as an (unused) keyword.
-------------	---

For example, here is a slightly enhanced version of the immutable `Deposit` class that we met in Chapter 5:

```

public final class Deposit implements Comparable<Deposit> {
    private final double amount;
    private final LocalDate date;
    private final Account payee;

    private Deposit(double amount, LocalDate date, Account payee) {
        this.amount = amount;
        this.date = date;
        this.payee = payee;
    }

    @Override
    public int compareTo(Deposit other) {
        return Comparator.nullsFirst(LocalDate::compareTo)
            .compare(this.date, other.date);
    }

    // methods elided
}

```

The immutability of this class rests upon the assumption that `Account` and all of its transitive dependencies are also immutable. This means, there are limits to what can be done - fundamentally the data model of Java and the JVM is not naturally friendly to immutability.

In the bytecode, we can see that finality fields shows up as a piece of field metadata:

```

$ javap -c -p out/production/resources/ch13/Deposit.class
Compiled from "Deposit.java"
public final class ch13.Deposit
    implements java.lang.Comparable<ch13.Deposit> {

    private final double amount;

    private final java.time.LocalDate date;

    private final ch13.Account payee;

    // ...
}

```

Trying to use immutable approaches to state in Java is bailing a leaking boat. Every single reference has to be checked for mutability, and if even one is missed then the entire object graph is mutable.

Worse still, the JVM's reflection and other subsystems also provide ways to circumvent immutability:

```

var account = new Account(100);
var deposit = Deposit.of(42.0, LocalDate.now(), account);
try {
    Field f = Deposit.class.getDeclaredField("amount");
    f.setAccessible(true);
    f.setDouble(deposit, 21.0);
    System.out.println("Value: " + deposit.amount());
} catch (NoSuchFieldException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
}

```

Taken together, all of this means that neither Java nor the JVM is an environment that provides any particular support for programming with immutable data. Languages like Clojure, which have stronger requirements, end up having to do a lot of the work in their language-specific runtime.

14.2.3 Higher-order functions

The concept of a higher-order function should not be surprising to a Java programmer. We have already seen an example of a static method `makePrefixer()` that takes in a prefix String and returns a function object.

Let's rewrite the code and change the static factory into another function object:

```

Function<String, Function<String, String>> prefixer =
    prefix -> s -> prefix +": "+ s;

```

This can be a little hard to read at first glance, so let's include some extra bits of syntax that we don't actually need, to make what's going on clearer:

```

Function<String, Function<String, String>> prefixer = prefix -> {
    return s -> prefix +": "+ s;
};

```

In this expanded view, we can see that `prefix` is the argument to the function and the returned value is a lambda (actually a Java closure) that implements `Function<String, String>`.

Notice the appearance of the function type `Function<String, Function<String, String>>` - it has two type parameters that define the input and output types. The second (output) type parameter is just another type - in this case, it is another function type. This is one way to recognize a higher-order function type in Java - a `Function` (or other functional type) that has `Function` as one of its type parameters.

Finally, we should point out that language syntax does matter - after all, function objects can be created as anonymous implementations, like this:

```

public class PrefixerOld
    implements Function<String, Function<String, String>> {

    @Override
    public Function<String, String> apply(String prefix) {
        return new Function<String, String>() {
            @Override
            public String apply(String s) {
                return prefix +": "+ s;
            }
        };
    }
}

```

This code would even have been legal as far back as Java 5, if the `Function` type had existed back then (and as far back as Java 1.1 if we remove the annotations). But it's a total eyesore - it's very hard to see the structure, which is why many programmers only think of functional programming as arriving with Java 8.

14.2.4 Recursion

The `javac` compiler provides a straightforward translation of Java source code into bytecode. As we can see, this applies to recursive calls:

```

public static long simpleFactorial(long n) {
    if (n <= 0) {
        return 1;
    } else {
        return n * simpleFactorial(n - 1);
    }
}

```

which compiles to the bytecode:

```

public static long simpleFactorial(long);
Code:
 0: lload_0
 1: lconst_0
 2: lcmp
 3: ifgt           8
 6: lconst_1
 7: lreturn
 8: lload_0
 9: lload_0
10: lconst_1
11: lsub
12: invokestatic #37           // Method simpleFactorial:(J)J
15: lmul
16: lreturn

```

This, of course, has some major limitations. In this case, making a call like `simpleFactorial(100000)` will result in a `StackOverflowError`. This is because of the `invokestatic` call at byte 12, which will cause an additional interpreter frame to be placed on the stack for each recursive call.

NOTE

A recursive method is one that calls itself. A tail recursive method is one where the self-call is the last thing that the method does.

Let's try to find a way to see if the recursive call could be avoided. One approach is to rewrite the factorial code into a tail recursive form, which in Java we can do most easily with a private helper method.

```
public static long tailrecFactorial(long n) {
    if (n <= 0) {
        return 1;
    }
    return helpFact(n, 1);
}

private static long helpFact(long i, long j) {
    if (i == 0) {
        return j;
    }
    return helpFact(i - 1, i * j);
}
```

The entry method, `tailrecFactorial()` does not do any recursion, it merely sets up the tail-recursive call and hides the details of the more complex signature from the user. The bytecode for the method is basically trivial, but let's include it for completeness.

```
public static long tailrecFactorial(long);
Code:
 0: lload_0
 1: lconst_0
 2: lcmp
 3: ifgt           8
 6: lconst_1
 7: lreturn
 8: lload_0
 9: lconst_1
10: invokestatic #49          // Method helpFact:(JJ)J
13: lreturn
```

As you can see, there are no loops and only a single branching `if` at bytecode 3. The real action (and the recursion) happens in `helpFact()`. This is still compiled by `javac` into bytecode which contains a recursive call, as we can see:

```

private static long helpFact(long, long);
Code:
 0: lload_0
 1: lconst_0
 2: lcmp
 3: ifne           8
 6: lload_2
 7: lreturn
 8: lload_0
 9: lconst_1
10: lsub
11: lload_0
12: lload_2
13: lmul
14: invokestatic #49 // Method helpFact:(JJ)J
17: lreturn

```

①

②

③

- ① Longs are 8 bytes, so need 2 local variable slots each
- ② Return from the $i == 0$ path
- ③ Tail-recursive call

In this form, however, we can now see that there are two paths through this method. The simple, $i == 0$ path starts at bytecode 0, falls through the if condition at 3 and returns j . The more general case is 0 to 3, then 8 to 14, which triggers a recursive call.

So on the only path that has a method call on it, the call is recursive, and always the last thing that happens before the `return` - i.e. the call is in *tail position*.

However, it *could* be compiled into this bytecode instead, which avoids the call:

```

private static long helpFact(long, long);
Code:
 0: lload_0
 1: lconst_0
 2: lcmp
 3: ifne           8
 6: lload_2
 7: lreturn
 8: lload_0
 9: lconst_1
10: lsub
11: lload_0
12: lload_2
13: lmul
14: lstore_2
15: lstore_0
16: goto          0

```

①

②

③

③

④

- ① Longs are 8 bytes, so need 2 local variable slots each
- ② Return from the $i == 0$ path
- ③ Reset the local variables
- ④ Jump to the top of the method

Now for the bad news - `javac` does not perform this operation automatically, despite it being

possible. This is yet another example of how the compiler tries to translate Java source into bytecode as exactly as possible.

NOTE

In the `resources` project that accompanies this book is an example of how to use the ASM library to generate a class that implements the above bytecode sequence - as `javac` will not emit it from recursive code.

For completeness, we should say that in practice, implementing a factorial function that handles longs with recursive calls rather than overwriting frames is not actually going to cause a problem, because the factorial increases so quickly that it will overflow the available space in a `long` well before any stack size limits are reached:

```
$ java TailRecFactorial 20
2432902008176640000

$ jshell
jshell> 2432902008176640000L + 0.0
$1 ==> 2.43290200817664E18

jshell> Long.MAX_VALUE + 0.0
$2 ==> 9.223372036854776E18
```

So `factorial(21)` is already larger than the largest positive `long` that the JVM can express.

However, while this specific trivial example is safe, it does not alter the difficult fact that all recursive algorithms in Java are potentially vulnerable to stack overflow.

This specific flaw is one of the Java language - and not of the JVM. Other languages on the JVM can, and do, handle this differently - for example by use of an annotation or a keyword. We will see examples of this when we discuss how Kotlin and Clojure handle recursion later in the chapter.

14.2.5 Closures

As we've already seen - a closure is essentially a lambda expression that captures some visible state from the scope in which the lambda is declared. Like this:

```
int i = 42;
Function<String, String> f = s -> s + i;
// i = 37;
System.out.println(f.apply("Hello "));
```

When this runs, it produces, as expected: `Hello 42`. However, if we uncomment the line that reassigned the value of `i`, then something different happens - the code stops compiling at all.

To understand why this happens, let's take a look at the bytecode that the code is compiled into. As we'll see in Chapter 16, the bodies of lambda expressions in Java are turned into private static methods. In this case, the lambda body turns into this:

```

private static java.lang.String lambda$main$0(int, java.lang.String);
Code:
 0: aload_1
 1: iload_0
 2: invokedynamic #32,  0 // InvokeDynamic #1:makeConcatWithConstants:
                           // (Ljava/lang/String;I)Ljava/lang/String;
 7: areturn

```

The clue is in the signature of `lambda$main$0()` - it takes **two** parameters, not one. The first parameter is the value of `i` that is passed in - which is 42 at the time that the closure is created (the second is the `String` parameter that the lambda takes when executed).

This means that Java closures contain copies of *values* - which are bit patterns (whether of primitives or object references) - and not *variables*.

NOTE **Java is strictly a pass-by-value language - there is no way in the core language to pass-by-reference or pass-by-name.**

This means that to see the effect of changes to captured state outside the scope of the closure body (or to effect things in other scopes), then the captured state must be a mutable object, like this:

```

var i = new AtomicInteger(42);
Function<String, String> f = s -> s + i.get();
i.set(37);
// i = new AtomicInteger(37);
System.out.println(f.apply("Hello "));

```

①
②

- ① Reassigning a value to mutable object state works
- ② This would fail to compile

In fact, in earlier versions of Java, only variables that were explicitly marked as `final` could have their value captured by Java closures. However, from Java 8 onwards, that restriction was changed to variables that are *effectively final* - variables that are used as though they were final, even if they don't actually have the keyword attached to their declaration.

This is actually symptomatic of a deeper problem. The JVM has a shared heap, method-private local variables and a method-private evaluation stack, and that's it. As compared to other languages, neither the JVM nor the Java language has the concept of an *environment* or a symbol table, or the ability to pass a reference to an entry in one.

Non-Java languages on the JVM that do have those concepts are required to support them in their language runtime - as the JVM does not provide any intrinsic support for them.

Some programming language theorists therefore reach the conclusion that what Java provides are not actually true closures, because of the additional level of indirection required. A Java

programmer must mutate the state of an object value, rather than being able to change the captured variable directly.

14.2.6 Laziness

Java does not provide first class support for lazy evaluation in the core language for ordinary values. However, one interesting place where we can see lazy evaluation in use is within the Java Streams API. Appendix B has a refresher on aspects of streams, should you require one.

NOTE

Laziness does play a role in some parts of the JVM and its programming environment (for example, aspects of classloading are lazy).

Calling `stream()` on a Java collection produces a `Stream` object, which is effectively a lazy representation of an ensemble of elements.

NOTE

Some streams can also be represented as a Java collection, however streams are more general and not every stream can be represented as a collection.

Let's look again at a typical Java `filter()` and `map()` pipeline:

```
stream()    filter()    map()    collect()
Collection -> Stream -> Stream -> Stream -> Collection
```

The `stream()` method returns a `Stream` object. The `map()` and `filter()` methods (like almost all of the operations on `Stream`) are lazy. At the other end of the pipeline, we have a `collect()` operation, which *materialises* the contents of the remaining `Stream` back into a `Collection`. This *terminal* method is eager, so the complete pipeline behaves like this:

```
lazy      lazy      lazy      eager
Collection -> Stream -> Stream -> Stream -> Collection
```

This means that, apart from the materialisation back into the collection, that the platform has complete control over how much of the stream to evaluate. This opens the door to a range of optimisations that are not available in purely eager approaches.

It can sometimes be helpful to think of the lazy, functional mode of Java streams as being analogous to hyperspace travel in a science fiction movie. Calling `stream()` is the equivalent of jumping from "normal space" into a hyperspace realm where the rules are different (functional and lazy, rather than OO and eager).

At the end of the operational pipeline, a terminal stream operation jumps us back from the lazy functional world into "normal space" - either by re-materializing the stream into a `Collection` (e.g. via `toList()`) or by aggregating the stream, via a `reduce()` or other operation.

Use of lazy evaluation does require more care from the programmer, but this burden largely falls upon library writers, e.g. the JDK developers.

However, there are some aspects of the lazy nature of streams that a Java developer should be aware of and respect the rules of. For example, implementations of some of the `java.util.function` interfaces (e.g. `Predicate`, `Function`) should not mutate internal state or cause side-effects. Violating this assumption can cause major problems if developers write implementations or lambdas that do.

Another important aspect of streams is that the stream objects themselves (the instances of `Stream` that are seen as intermediate objects within a pipeline of stream calls) are single-shot. Once they have been traversed they should be considered as invalid. In other words, developers should not attempt to store or re-use a stream object, as the results of doing so are almost certainly incorrect and attempts may throw.

NOTE

Placing a stream object into a temporary variable is almost always a code smell - although doing so during development when debugging a complex generics issue with a stream is acceptable, provided the use of stream temporaries is removed when the code is completed.

One other aspect of the laziness of streams is the ability to model more general data than collections. For example, it is possible to construct an infinite stream by use of `Stream.generate()` combined with a generating function. Let's take a look:

```
public class DaySupplier implements Supplier<LocalDate> {
    private LocalDate current = LocalDate.now().plusDays(1);

    @Override
    public LocalDate get() {
        var tmp = current;
        current = current.plusDays(1);
        return tmp;
    }

    final var tomorrow = new DaySupplier();
    Stream.generate(() -> tomorrow.get())
        .limit(10)
        .forEach(System.out::println);
```

This produces a stream of days that is infinite (or as large as is needed, if you prefer). This would be impossible to represent as a collection, without running out of space, thus showing that streams are more general.

This example also shows that Java's restrictions, such as pass-by-value, restrict the design space somewhat. The `LocalDate` class is immutable, and so we are required to have a class containing a mutable field `current` and then to mutate `current` within the `get()` method in order to provide a stateful method that can generate a sequence of `LocalDate` objects.

In a language that supported pass-by-reference, the type `DaySupplier` would have been unnecessary, as `current` could have been a local variable declared in the same scope as `tomorrow`, which could then have been a lambda.

14.2.7 Currying and partial application

We already know that Java does not have any language-level support for currying, but we can take a quick look at how something could have been added. For example, here is the declaration for the `BiFunction` interface in `java.util.function`:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);

    default <V> BiFunction<T, U, V> andThen(
        Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t, U u) -> after.apply(apply(t, u));
    }
}
```

Notice how the default methods feature of interfaces is used to define `andThen()` - an additional method, beyond the standard `apply()` method for the `BiFunction`. This same technique could have been used to provide some support for currying, for example by defining two new default methods:

```
default Function<U, R> curryLeft(T t) {
    return u -> this.apply(t, u);
}

default Function<T, R> curryRight(U u) {
    return t -> this.apply(t, u);
}
```

These define two ways to produce Java `Function` objects - i.e. functions of one argument from our original `BiFunction`. Notice that they are implemented as closures - we're simply capturing the supplied value and storing it for later, when we actually apply the function.

We could then use these additional default methods like this:

```
BiFunction<Integer, LocalDate, String> bif =
    (i, d) -> "Count for " + d + " = " + i;

Function<LocalDate, String> withCount = bif.curryLeft(42);
Function<Integer, String> forToday = bif.curryRight(LocalDate.now());
```

However, the syntax is somewhat clunky - it requires two different methods for the two possible curries, and they must have different names due to type erasure. Even after all that, the resulting feature is arguably of only limited use. So this approach was never implemented and, as discussed, Java does not support currying out of the box.

14.2.8 Java's type system and collections

To conclude our unfortunate tale about Java's less-than-great affinity to functional programming, let's talk about Java's type system and collections. There are three main issues with these parts of the Java language that contribute to the somewhat poor fit to the functional programming style:

- Non-single-rooted type system
- `void`
- Design of the Java Collections

First of all, Java has a non-single rooted type system (i.e. there is no common supertype of `Object` and `int`). This makes it impossible to write `List<int>` in Java, and as a result leads to autoboxing and the attendant problems.

NOTE

Lots of developers complain about the erasure of type parameters of generic types during compilation, but in reality it is more often the non-single-rooted type system that really causes problems with generics in the Collections.

Java has another problem, connected to the non-single-rooted type system: `void`. This keyword indicates that a method does not return a value (or, looked at another way, that the evaluation stack of the method is empty when the method returns). The keyword therefore carries the semantics that whatever the method does, it is acting purely by side-effect - it's "the opposite of pure" in some sense.

The existence of `void` means that Java has both statements and expressions and that it is impossible to implement the design principle "everything is an expression" which some functional programming traditions are very keen on.

Another problem is related to the shape and nature of the Java Collections interfaces. They were added to the Java language with version 1.2 (aka Java 2), which was released in December 1998. They were not designed with functional programming in mind.

A major problem for doing FP with the Java Collections is that an assumption of mutability is built in everywhere. The collections interfaces are large and explicitly contain methods like these from `List<E>`:

- `boolean add(E e)`
- `E remove(int index)`

These are mutation methods - the signature of them implies that the collection object itself has been modified in-place.

The corresponding methods on an immutable list would have signatures such as `List<E> add(E`

`e`), that return a new, modified copy of the list. The case of `remove()` would be difficult to implement, because Java doesn't have the ability to return multiple values from a method.

NOTE

The real problem is that `remove()` is incorrectly factored for FP - a very similar case to that of the `Iterator` that we discussed in Section 10.4

All of this therefore implies that **any** implementation of the collections is implicitly expected to be mutable. There does exist the horrible hack of using `UnsupportedOperationException`, which we discussed in Section 6.5.3, but this is not something that a well-grounded Java developer should use.

Other, non-Java languages, separate out the concept of the collection type from mutability, e.g. by representing them as different interfaces (or different *traits* in languages that support that concept). This enables implementations to specify whether or not they are mutable at type level, by choosing to implement, or not, the separate interfaces.

Lurking behind all of this is that one of Java's main virtues and most important design principles is backwards compatibility. This makes it difficult, or impossible, to change some of these aspects to make the language more functional.

For example, in the case of the collections, rather than try to add additional, functional methods directly onto the collections interfaces, a clean break was made and `Stream` was introduced, to act as a new container type that did not have the implicit semantics of the collections.

Of course, just introducing a new container type and API does nothing to change the millions upon millions of lines of existing code that uses the collections. Nor does it help in the slightest for the common case where an API is already expressed in terms of a collection type.

NOTE

This problem is not unique to the stream / collection divide. For example, Java Reflection was introduced in Java 1.1, and predates the arrival of the collections. As a result, the API is annoyingly difficult to use, as it relies upon arrays as element containers.

This section has shown some rather depressing facts about the state of support for functional programming in Java. The takeaway message is that simple functional patterns (such as Filter-Map-Reduce) are available. These are very useful for all sorts of applications as well as generalizing well to concurrent (and even distributed) applications - but they are about the limit of what Java is able to do.

Let's move on look at our non-Java languages and see if the news is any better.

14.3 Kotlin FP

We've already demonstrated how modern Java handles some basic, common patterns in the functional programming paradigm. It probably comes as no surprise that Kotlin brings conciseness and a few additional ideas to the table for the functionally inclined.

14.3.1 Pure and higher-order functions

Back in section 9.2.4 we introduced Kotlin's functions. In Kotlin functions are part of the type system, expressed with syntax like `(Int) → Int` where the contents of the parenthesized list are the argument types, and the right of the arrow is the return type.

By using this notation, we can easily write down signatures for functions which accept other functions as arguments or return a function - i.e. *higher-order functions*. Kotlin naturally encourages the use of such higher-order functions. Much of the API around working with collections such as `map` and `filter` are in fact built off these higher-order functions, just as we see in the Java (and Clojure) APIs that provide the equivalent language feature.

But higher-order functions aren't restricted to collections and streams. For instance, here's a classic functional programming function called `compose`. `compose` will return a function which calls each of the functions passed as arguments to it.

```
fun compose(callFirst: (Int) -> Int,
           callSecond: (Int) -> Int): (Int) -> Int {
    return { callSecond(callFirst(it)) } ①
}

val c = compose({ it * 2 }, { it + 10 }) ②
c(10) ③
```

- ① `compose` returns a function, so `callFirst` and `callSecond` aren't called when this line executes
- ② We pass two lambdas, using the `it` shorthand described in Chapter 9 to avoid explicitly listing the single argument to the lambdas.
- ③ We invoke and run the function returned by `compose`, which returns 30.

Kotlin provides a number of ways to get a handle to a function depending on your needs. You can declare lambda expressions as shown above (and with many other flavors and features discussed in chapter 9).

Alternatively, we can refer to a named function via the `::` syntax.

```
fun double(x: Int): Int {
    return x * 2
}

val c = compose(::double, { it + 10 }) ①
c(10)
```

- ① Same result as our prior example

:: knows more than just top-level functions. It can also refer to a function belonging to a specific object instance.

```
data class Multiply(var factor: Int) {
    fun apply(x: Int): Int = x * factor
}

val m = Multiply(2)
val c = compose(m::apply, { it + 10 })      ①
c(10)
```

- ① References the `apply` method on the `Multiply` class bound specifically to our instance `m`.

Sadly, much like Java, Kotlin provides no built-in way of guaranteeing the purity of a given function. While defining functions at the top-level (outside of any class) and using `val` to ensure immutable data can take you a long way, they don't ensure the referential transparency of your functions.

14.3.2 Closures

An aspect of lambda expressions which may not be obvious on the surface is how they interact with the surrounding code. For instance, the following code works even though `local` isn't declared within our lambda.

```
var local = 0
val lambda = { println(local) }      ①
lambda()
```

- ① Prints 0

This is referred to as *closure* (as in, the lambda *closes over* the values it can see). Importantly, and unlike Java, it is not just the value of the variables the lambda can access - under the covers it actually keeps a reference to the variables themselves.

```
var local = 0
val lambda = { println(local) }      ①
lambda()

local = 10
lambda()                            ②
```

- ① Prints 0
 ② Prints 10, the updated value of `local` at the time `lambda` is invoked.

This closure over variables remains even if the variables themselves would otherwise have gone out of scope. Here we return a lambda from a function, keeping a reference to a variable that

normally would be inaccessible.

```
fun makeLambda(): () -> Unit {
    val inFunction = "I'm from makeLambda"      ①
    return { println(inFunction) }
}

val lambda = makeLambda()
lambda()                                ②
```

- ① `inFunction` would normally go out of scope when `makeLambda` is done
- ② Because our lambda expression closes over `inFunction`, it is still available here - but only inside our lambda.

NOTE **Lambdas carrying references outside their typical scope can be a source for unexpected object leaks!**

The location of a lambda expression's declaration determines what it may capture in its closure. For instance, if declared within a class, then the lambda can close over properties in the object

```
class ClosedForBusiness {
    private val amount = 100
    val check = { println(amount) }      ①
}

fun getTheCheck(): () -> Unit {
    val closed = ClosedForBusiness()
    return closed.check                 ②
}

val check = getTheCheck()
check()                                ③
```

- ① Lambda saved into `check` closes over the private property `amount`
- ② This function returns that lambda, which keeps a reference to `amount`. This keeps the instance `closed` alive when normally it wouldn't exist after the function completes.
- ③ Prints 100 when called. When the `check` variable exits scope, the `closed` instance will also finally be eligible for garbage collection.

Closures with higher-order functions provide a rich basis for building new functions from old ones.

14.3.3 Currying and partial application

The story for Kotlin is very similar to that in Java. Let's look at an example:

```

fun add(x: Int, y: Int): Int {
    return x + y
}

fun partialAdd(x: Int): (Int) -> Int {
    return { add(x, it) }
}

val addOne = partialAdd(1)
println(addOne(1))
println(addOne(2))

val addTen = partialAdd(10)
println(addTen(1))
println(addTen(2))

```

This is really just a syntactic trick that works because the () operator desugars to a call to the `apply()` method. At bytecode level this is really just the same as the Java example. We could imagine some helper syntax for automatically creating curries, perhaps something like:

```

val addOne: (Int) -> Int = add(1, _)
println(addOne(10))

```

However, the core language does not directly support this. Various third-party libraries can provide similar, slightly more verbose abilities, often via an extension method.

14.3.4 Immutability

Section 14.1.2 framed immutability as a key technique for success in functional programming. If a pure function returns data which is meant to be identical for a given input, allowing an object to change after the fact breaks the guarantees that purity bought us.

A primary feature of Kotlin that aids in our quest for immutability is the `val` declaration. `val` ensures a property may only be written during object construction, much like Java's `final`. In fact, `val` is effectively the same as Java's `final var` combination, but is also applicable to properties and much less awkward to write.

Chapter 9 covers the many locations where Kotlin supports use of `val/var`, but to succeed in functional programming it's recommended to embrace immutability and favor `val` over `var`. Kotlin's built-in support for properties also sweeps away the getter boilerplate required in Java.

```

class Point(val x: Int, val y: Int)

val point = Point(10, 20)
println(point.x)
// point.x = 20 // Won't compile because x is immutable!

```

A major hangup with immutable objects, though, is the difficulty when you actually want to change something. To retain our immutability, we must create entirely new instances, but this can be tedious and error-prone. In Java this is often tackled with static factory methods, builder objects, or *wither* method to cut down on the noise.

Kotlin's `data class` construct gives us a nice alternative to those approaches. In addition to the constructor and equality operations we covered in section 9.3.1, a data class also gets a `copy` method. Pairing `copy` with Kotlin's named arguments, you can generate the new instance we want, only writing out the changes you actually want.

```
data class Point(val x: Int, val y: Int)

val point = Point(10, 20)
val offsetPoint = point.copy(x = 100)
```

`copy` does come with a couple important caveats. First, it is a shallow copy. If one of our fields is an object, we copy the reference to that object, not the full object itself. Like in Java, if any of the chain of objects allows mutation, then our guarantees are broken. For true immutability, all the objects involved need to play along but the language will not enforce it for you.

Another point of caution is that `copy` is generated based off the constructor of the class alone. If we bend our rules and put `var` fields elsewhere on our objects, `copy` doesn't know about these additional fields and they get default values only in any copy.

```
data class Point(val x: Int, val y: Int) {
    var shown: Boolean = false
}

val point = Point(10, 20)
point.shown = true

val newPoint = point.copy(x = 100)
println(newPoint.shown)
```

①

- ① Outputs `false`, because the non-constructor fields aren't touched by `copy`

But we'd never let a mutable field sneak into our nice immutable objects to begin with, right?

Controlling the mutability of our objects is an important first step, but most non-trivial code will involve collections of objects, not just individual instances. We saw in Chapter 9 that Kotlin's functions for constructing collections (e.g. `listOf`, `mapOf`) return interfaces such as `kotlin.collections.List` and `kotlin.collections.Map` which unlike their counterparts in `java.util` are read-only. Sadly while this is a good start, it doesn't provide us the guarantees we want.

We can't trust the immutability of these objects because the mutable interfaces extend the read-only flavors. Anywhere you can pass a `List`, you can pass a `MutableList`.

```
fun takesList(list: List<String>) {
    println(list.size)
}

val mutable = mutableListOf("Oh", "hi")
takesList(mutable)

mutable.add("there")
takesList(mutable)
```

`takesList` receives the same object in both invocations, but the result of the call is different. Our functionality purity is shattered!

NOTE

The read-only helpers like `listOf` use underlying JDK collections and return objects which are read-only. For instance, `listOf` defaults to an array-backed list implementation which cannot be added to. It's just the mix of Kotlin's mutable interfaces with the standard read-only interfaces that spoil the party.

Implementing these collections via JDK classes also leaves some sharp edges if you cast between interfaces. Kotlin's aim for clean interop with Java collections means the result from `listOf()` can be cast to both Kotlin's mutable interfaces, and the classic `java.util.List<T>` where we can attempt to modify the collection! The following code compiles without a complaint, but fails at runtime.

```
fun takesMutableList(muted: MutableList<Int>) {
    muted.add(4)
}

val list = listOf(1,2,3)
takesMutableList(list as MutableList<Int>)
```

①

① This call will throw a `java.lang.UnsupportedOperationException`.

This lack of real immutability becomes problematic especially when we're talking about concurrency. As we saw in Chapter 6, making mutable collections safe between multiple threads takes quite an effort. If we had a collection instance that was truly immutable, though, that could be freely distributed among different threads of execution, assured that everyone is getting an identical picture of the world.

While Kotlin doesn't have them in the standard library, the `kotlinx.collections.immutable` library provides a variety of immutable and *persistent* data structures. Frequently seen libraries such as Guava and Apache Commons also have many similar options.

What does it mean for a collection to be persistent? As we've discussed multiple times, immutability means that when you need to "change" an object, you instead create a new instance of it. For large collections this could be really inefficient. Persistent collections lean on immutability to lower that cost of modification - they are built to safely share immutable parts of

their internal storage. While you still create a new object to effect any change, those new objects can be much smaller than a full copy.

```
import kotlinx.collections.immutable.persistentListOf

val pers = persistentListOf(1, 2, 3)
val added = pers.add(4)
println(pers)           ①
println(added)          ②
```

- ① Prints [1, 2, 3]
- ② Prints [1, 2, 3, 4]

The core of the library implements two groups of interfaces typified by `ImmutableList` and `PersistentList`. Matching pairs exist for maps, sets and general collections as well. `ImmutableList` extends `List`, but unlike its base interface, guarantees any instance is immutable. `ImmutableList` can then be used in any locations where you're passing lists and want to enforce immutability. `PersistentList` builds off of `ImmutableList` to provide us with the "modify and return" methods.

The library also includes familiar extensions for converting other collections into persistent versions.

```
val mutable = mutableListOf(1,2,3)
val immutable = mutable.toImmutableList()
val persistent = mutable.toPersistentList()
```

At this point, you might be wondering why we don't change every `listOf` to `persistentListOf` "just in case." But these aren't the default implementation for a reason. While persistent data structures lower the cost of copies, they still can't touch the speed of classic mutable data structures. Less copying *does not* equal no copying! How much does this cost?

As Chapter 7 has hopefully convinced you, the only way to know is to measure in your own use cases. But if you need concurrent access to a collection across threads, it's worth comparing how these persistent structures perform versus more standard copying with synchronization.

Now that we have Kotlin's toolkit for making data immutable in hand, let's look at a feature it brings to the world of recursive functions.

14.3.5 Tail recursion

In section 14.2.4 we examined recursive functions in Java. They have a major limitation as each successive recursive function call adds a stack frame which eventually exhausts the available space. Kotlin has the same limitation with basic recursion as we can see from the translation of our `simpleFactorial` function to Kotlin (note the use of a Kotlin `if` expression as the return value).

```
fun simpleFactorial(n: Long): Long {
    return if (n <= 0) {
        1
    } else {
        n * simpleFactorial(n - 1)
    }
}
```

This yields the following bytecode, which is equivalent to what `javac` emitted for our Java function.

```
public final long simpleFactorial(long);
Code:
 0: lload_1
 1: lconst_0
 2: lcmp
 3: ifgt      10
 6: lconst_1
 7: goto      22
10: lload_1
11: aload_0
12: checkcast     #2                  // class Factorial
15: lload_1
16: lconst_1
17: lsub
18: invokevirtual #19                // Method simpleFactorial:(J)J
21: lmul
22: lreturn
```

Apart from a little additional validation (byte 12) and the use of a `goto` instead of multiple `lreturn` instructions, this is fundamentally the same. The recursive call at byte 18 where we `invokevirtual` on `simpleFactorial` will eventually blow the stack.

```
java.lang.StackOverflowError
  at Factorial.simpleFactorial(factorial.kts:32)
  at Factorial.simpleFactorial(factorial.kts:32)
  at Factorial.simpleFactorial(factorial.kts:32)
...
...
```

While this problem is unavoidable in the general case, Kotlin can help us out if our function is tail recursive.

Remember that a tail recursive function is one where the recursion is the last operation in the entire function. Earlier, we showed how at the bytecode level we could reset state and `goto` the top of the function instead of adding a stack frame. This transforms our recursive call into a loop, with no danger of overflowing the stack. Java didn't provide us any way to do this, but Kotlin does.

NOTE Any recursive function can be rewritten to be tail-recursive. It may require additional parameters, variables, and tricks to do the transformation, but it is always possible. Given that tail-recursive functions can be transformed into simple looping, this shows that any recursive function can also be implemented iteratively using only loop constructs.

To get our factorial recursive call into the final position takes a little shuffling. As in Java, we split the function to retain the nice single argument form for users, and put the more complicated recursive function which now needs multiple arguments into a separate function.

```
fun tailrecFactorial(n: Long): Long {
    return if (n <= 0) {
        1
    } else {
        helpFact(n, 1)
    }
}

tailrec fun helpFact(i: Long, j: Long): Long { ①
    return if (i == 0L) {
        j
    } else {
        helpFact(i - 1, i * j)
    }
}
```

- ① Our helper is marked `tailrec` so Kotlin knows to look for tail recursion

The entry function `tailrecFactorial` doesn't hide any surprises in the bytecode. It does our beginning range check and then hands off to our tail recursive helper.

```
public final long tailrecFactorial(long);
Code:
 0: lload_1
 1: lconst_0
 2: lcmp
 3: ifgt      10
 6: lconst_1
 7: goto      19
10: aload_0
11: checkcast #2           // class Factorial
14: lload_1
15: lconst_1
16: invokevirtual #10       // Method helpFact:(JJ)J
19: lreturn
```

Bytes 0-3 check for an early return implemented by byte 6-7. If we need to make the recursive call, then it loads up the values needed to `invokevirtual` for `helpFact` at byte 16.

The important difference the `tailrec` keyword introduces shows up in the bytecode for `helpFact`.

```

public final long helpFact(long, long);
Code:
 0: lload_1
 1: lconst_0
 2: lcmp
 3: ifne      10
 6: lload_3
 7: goto     26
10: aload_0
11: checkcast #2           // class Factorial
14: pop
15: lload_1
16: lconst_1
17: lsub
18: lload_1
19: lload_3
20: lmul
21: lstore_3
22: lstore_1
23: goto     0
26: lreturn

```

The majority of this method is doing the logical checks and arithmetic of our factorial, but the key is at byte 23. Instead of doing an `invokevirtual` for `helpFact` to recurse, instead we simply `goto 0` and start the function again. With no invoke instructions present, we have no danger of stack overflows which is great news. Who says `goto` is always hazardous?

Tail recursion is an elegant solution when your function can be rewritten in the proper form. But what if you ask for it on a function which isn't tail recursive?

```

tailrec fun simpleFactorial(n: Long): Long { ①
    return if (n <= 0) {
        1
    } else {
        n * simpleFactorial(n - 1)
    }
}

```

- ① Inappropriately asking for `tailrec` when our final call isn't ourselves - in this case, the final operation is the `*` on the result of the recursive call

Kotlin spots the problem and warns us that it can't transform the bytecode to take advantage of tail recursion, pointing us straight to our incorrect, not-at-the-end call.

```

factorial.kts:28:1: warning: a function is marked as tail-recursive
                                but no tail calls are found

tailrec fun simpleFactorial(n: Long): Long {
^
factorial.kts:32:9: warning: recursive call is not a tail call
    n * simpleFactorial(n - 1)
^

```

Issuing a warning is not an especially strong behavior for the compiler - as it has the possibility that after a tail recursive implementation has been created, it can be subsequently subtly modified to *not* be tail recursive. Unless the build process flags the warning, this code can escape into

production and cause a `StackOverflowError` at runtime.

Arguably, it would be better if declaring a non-tail-recursive function as `tailrec` caused a compilation error, as is done in some other languages (e.g. Scala).

14.3.6 Lazy evaluation

As we mentioned earlier in the chapter, many functional languages (such as Haskell) rely heavily on *lazy evaluation*.

As a language on the JVM Kotlin doesn't center lazy evaluation in its core execution model. But it does bring first-class support for laziness where you want it via the `Lazy<T>` interface. This provides a standard structure for when you want to delay - or potentially skip entirely - a bit of processing.

Typically you don't implement `Lazy<T>` yourself, but instead use the `lazy()` function to construct instances. In the simplest form, `lazy()` takes a lambda, the return type of which determines the type `T` of the returned interface.

```
val lazing = lazy {
    42
}

println("init? ${lazing.isInitialized()}")      ①
println("value = ${lazing.value}")               ②
println("init? ${lazing.isInitialized()}")        ③
```

- ① Check whether we're initialized, will report `false`
- ② Access to `value` will force our lambda to execute and save the result
- ③ Check whether we're initialized, will report `true`

Our desire to put off unneeded computation may overlap with the need to execute across multiple threads. When that happens, `lazy()` takes a `LazyThreadSafetyMode` enumeration to help control how that happens. The enum values are `SYNCHRONIZED` (the default for `lazy()`), `PUBLICATION`, `NONE`.

- `SYNCHRONIZED` uses the `Lazy<T>` instance itself to synchronize execution of the initializing lambda.
- `PUBLICATION` instead allows multiple concurrent executions of the initialization lambda, but saves only the first value seen.
- `NONE` skips synchronization, with undefined behavior if accessed concurrently.

NOTE

`LazyThreadSafetyMode.NONE` should only be used if you've 1) measured that synchronization in your lazy instances is an actual performance problem and 2) can somehow guarantee you'll never access the object from multiple threads. The other options, `SYNCHRONIZED` and `PUBLICATION`, can be chosen between based on whether your use-case is sensitive to the initialization lambda running multiple times concurrently.

The `Lazy<T>` interface is designed to work with another advanced Kotlin feature called *delegated properties*. When defining a property on a class, instead of providing the value or a custom getter/setter, you can instead provide an object with the `by` keyword. That object must have an implementation of `getValue()` and (for `var` properties) `setValue()`. `Lazy<T>` fits this specification so we can easily put off initializing properties in our classes without repeating boilerplate or veering away from natural Kotlin syntax.

```
class Procrastinator {
    val theWork: String by lazy {
        println("Ok, I'm doing it...") ①
        "It's done finally"
    }
}

val me = Procrastinator()

println(me.theWork) ②
println(me.theWork)
println(me.theWork) ③
```

- ① Diagnostic message to make it easier to prove everything works
- ② First call to `theWork` will run the lambda and print the working message
- ③ Further calls to `theWork`, as seen on the two concluding lines, will just return the same, already calculated value

Much like immutability, laziness is excellent for our own objects, but leaves us wondering about collections and iteration. Next we'll look at how Kotlin allows us to better control the flow of execution when streaming through collections with the `Sequence<T>` interface.

14.3.7 Sequences

While Kotlin's collection functions are frequently convenient, they assume we will eagerly apply the function to the entire collection. An intermediate collection is created for each step in our chain of functions, potentially a waste if we don't actually need the whole result.

```
val iter = listOf(1, 2, 3)
val result = iter
    .map { println("1@ $it"); it.toString() } ①
    .map { println("2@ $it"); it + it } ②
    .map { println("3@ $it"); it.toInt() } ③
```

- ① Generates an intermediate collection with ["1", "2", "3"]
- ② Generates an intermediate collection with ["11", "22", "33"]
- ③ Generates our final result of [11, 22, 33]

This outputs the following where we can see each step of our chain of `map` calls gets executed across the full list before the next `map` runs.

```
1@ 1
1@ 2
1@ 3
2@ 1
2@ 2
2@ 3
3@ 11
3@ 22
3@ 33
```

```
val iter = listOf(1, 2, 3)
val result = iter
    .map { it.toString() }
    .map { it + it }
    .map { it.toInt() }
```

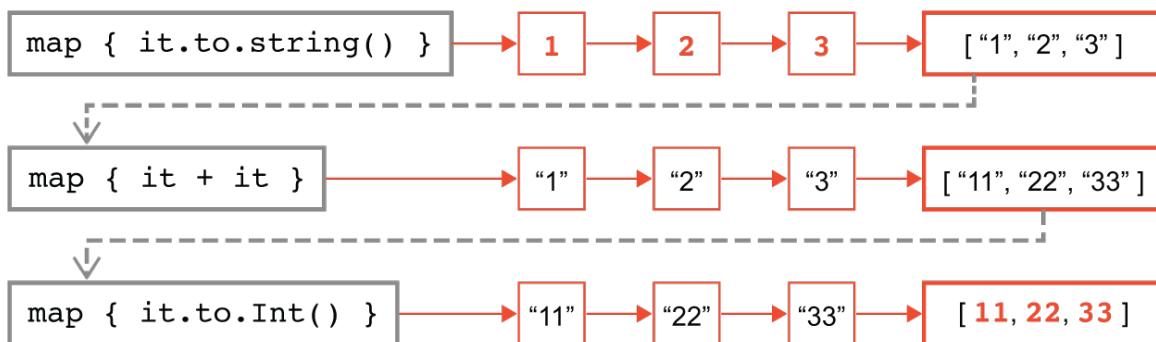


Figure 14.1 Standard iteration through collections

Beyond the possibility for wasted resources, there are also use-cases where our inputs are potentially infinite. What if we wanted to continue this mapping across as many numbers as we can until the user tells us to stop? We can't create the list ahead of time and process the full thing step by step.

To handle this, Kotlin has *sequences*. At the core of it is the interface `Sequence<T>` which looks similar to `Iterable<T>` but under the covers gives us a whole new set of capabilities.

We can create a new sequence using the `sequenceOf()` function and then start applying functions just like the collections we're used to. In the following example we've turned our list into a sequence and kept the diagnostic prints so we can see what's going on.

```
val seq = sequenceOf(1, 2, 3)
val result = seq
    .map { println("1@ $it"); it.toString() }      ①
    .map { println("2@ $it"); it + it }
    .map { println("3@ $it"); it.toInt() }
```

- ① Note that + here is string concatenation, not numeric addition

When we run this short program we'll find that nothing prints. This is because the primary feature of sequences is that they are lazy in their evaluation. Nothing in this program actually requires the return of our `map` calls so Kotlin just doesn't run them! If we turn the sequence into a list, though, the program will be forced to evaluate everything and we can see how the control flow of the sequence runs.

```
val seq = sequenceOf(1, 2, 3)
val result = seq
    .map { println("1@ $it"); it.toString() }
    .map { println("2@ $it"); it + it }
    .map { println("3@ $it"); it.toInt() }
    .toList()
```

This will have the following output:

```
1@ 1
2@ 1
3@ 11
1@ 2
2@ 2
3@ 22
1@ 3
2@ 3
3@ 33
```

We can see that each element of the sequence goes through the `map` chain individually, first 1, then 2, etc. before the following element is processed.

```
val seq = sequenceOf(1, 2, 3)
val result = seq
    .map { it.toString() }
    .map { it + it }
    .map { it.toInt() }
    .toList()
```

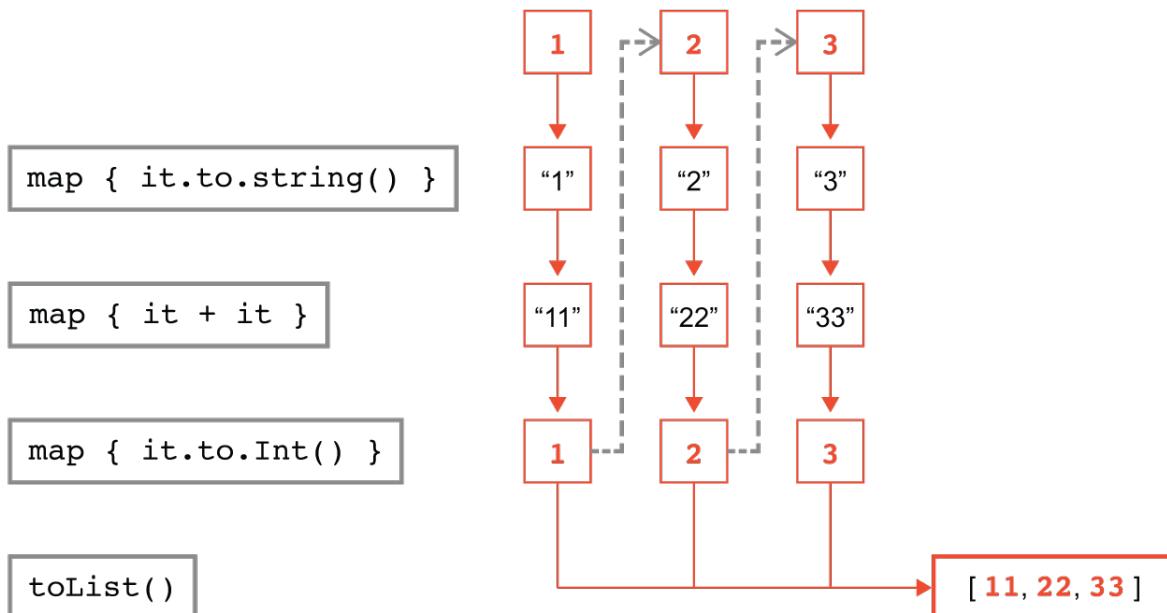


Figure 14.2 Sequence execution

This is interesting, but on relatively small, static lists probably not that compelling - proper measurement is deserved, but the bookkeeping that sequences require may well drown out the wins for not allocating intermediate collections. The power of sequences comes clearer with the alternative ways of creating sequences.

Our first stop is the `asSequence()` function. This will, unsurprisingly, turn things that are iterable into a sequence. This works with more than just the lists and collections you might expect, though, and may be called on *ranges*.

We met Kotlin's numeric ranges back in section 9.2.6 where they were used to check inclusion with `when` expressions. But ranges may be iterated across too. We can combine this with `asSequence()` to create long numeric lists without bothersome typing or over allocating.

```
(0..1000000000)      ①
    .asSequence()
    .map { it * 2 }
```

- ① Range only tracks its begin and end, so we don't create a billion elements

But what if even the large-but-still-bounded nature of ranges feels too restrictive?

`generateSequence()` from `kotlin.sequences` in the standard library has you covered. This function creates a new, general sequence object with an optional starting value. Each time it needs the next element, it runs the provided lambda, passing in the prior value.

```
generateSequence(0) { it + 2 } ①
```

- ① An infinite sequence of even numbers

An infinite `Iterable<T>` would be impossible to chain methods with as the first call on it would never return. `Sequence<T>` will just get what it needs and leave the rest for later. This pairs nicely with the `take()` function, where we can ask for a specific number of elements to be retrieved as a new, bounded sequence.

```
generateSequence(0) { it + 2 }
    .take(3) ①
    .forEach { println(it) } ②
```

- ① Create a sequence with the first three elements in it
- ② Force evaluation through the sequence and print what we received

`forEach()` on sequences is what's referred to as *terminal*, because it ends the sequence's laziness and evaluates everything it has. We've seen another terminal already with `toList()`, which necessarily steps through every element to construct a list.

Kotlin offers another option for creating a sequence if for some reason it's difficult to work from just the prior element in the sequence. The pairing of `sequence()` with `yield()` lets us construct completely arbitrary sequences.

```
val yielded = sequence {
    yield(1)
    yield(100)
    yieldAll(listOf(42, 3)) ①
}
```

- ① `yieldAll()` takes an iterable of the same type we're yielding and will `yield` each element in turn when asked.

As we'd expect with sequences, the lambda is lazily executed to determine the next element. What's unique here, though, is that for each call for the next element, the lambda only runs until the next `yield` and then pauses. A subsequent request for another element will resume the lambda where it had paused and run until the next `yield` again.

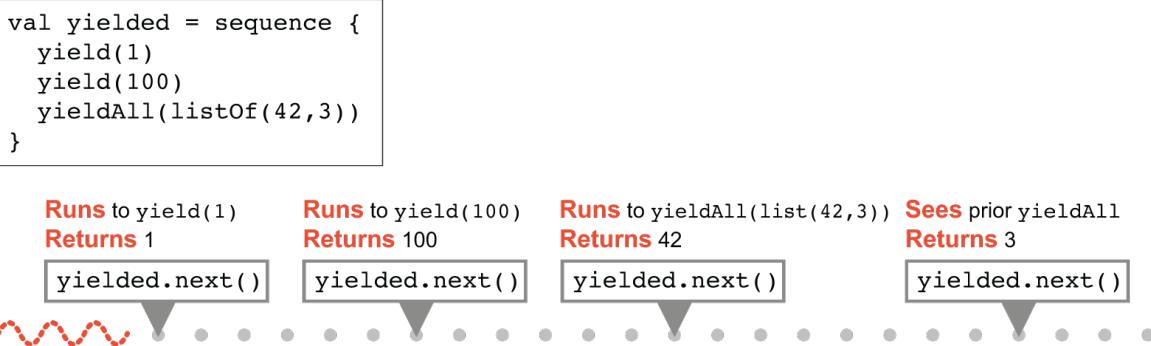


Figure 14.3 Timeline view of yield execution

`yield` uses a Kotlin feature known as *suspend* functions. As the name suggests, these are functions where Kotlin recognizes points in code execution that can stop and resume. In this case, Kotlin sees that each time we `yield` a value, execution our `sequence` lambda should pause until the next value is requested. While our code looks like a simple lambda, the Kotlin compiler is actually doing a lot of additional work for us behind the scenes.

Suspend functions are deeply related to Kotlin's alternative concurrency model, coroutines, which we introduced in section 9.4 and will discuss on in more detail in Chapter 15. It's an interesting point to note, though, that a feature that we often consider in light of concurrency also unlocks unique ways of functional programming as well.

14.4 Clojure FP

We met the basics of functional programming in Clojure in Chapter 10 - with forms like `(map)`. We also had early introductions to concepts like immutability and higher-order functions - because those ideas and capabilities are very close to the core of the Clojure programming model.

So, in this section, rather than introducing the Clojure take on the features we discussed for Java and Kotlin, we will be going beyond those foundations and showing how some of Clojure's more advanced functional features work, starting with a note on list comprehensions.

14.4.1 Comprehensions

Another important idea in functional programming is the concept of a *comprehension*, where the word means a "complete description" of a set or other data structure.

This concept is derived from mathematics, where we often see set-theoretic notation used to describe sets, like this:

```
{ x   : (x / 2) }
```

In this notation, means "is a member of", is the infinite set of all *natural numbers* (or counting numbers) - which are the numbers we would use to count objects (so 1, 2, 3, etc) and the `:` defines a condition or a *restriction*.

So this comprehension is describing a set of counting numbers that have a special property - every number in the set, when divided by two, yields a number that is also a counting number. Of course, we already know this set by another name - it is the set of *even numbers*.

The key point is that we did not specify the set of even numbers by listing out the elements (that would be impossible - there are an infinite number of them). Instead, we defined "even numbers" by starting from the natural numbers, and specifying an additional condition that had to hold for each element to be included in the new set.

If that sounds a bit like the use of functional techniques, such as `filter`, then it should - they are very closely related concepts. However, functional languages often offer both comprehensions and `filter-map` because they turn out to be conceptual easier to use in different circumstances.

Clojure implements *list comprehensions* using the `(for)` form to return either a list (or an iterator in some cases). This is why, when we meet Clojure loops in Chapter 10, that we didn't introduce `(for)` - because it isn't really a loop.

Let's see it in action:

```
user=> (for [x [1 2 3]] (* x x))
(1 4 9)
```

The `(for)` form takes two arguments - an argument vector, and a form that will represent the values to *yield* as part of the overall list that `(for)` will return.

The argument vector contains a pair (or multiple pairs) of elements - a temporary that will be used in the definition of the yielded values and a seq to provide the inputs. We can think of the temporary as being used to bind each of the values, in turn.

This could, of course, easily be written as a map:

```
user=> (map #(* % %) [1 2 3])
(1 4 9)
```

So, where would we want to use `(for)`? The answer is that it comes into its own when we have more complex structures to build up, for example:

```
(for [num [1 2 3]
      ch [:a :b :c]]
  (str num ch))
("1:a" "1:b" "1:c" "2:a" "2:b" "2:c" "3:a" "3:b" "3:c")
```

This could also be done with a map, but the construction would potentially be more complex and

cumbersome, whereas with `(for)` it is clear and straightforward.

To get the effect of a filter, we can also use an additional qualifier on the `(for)` that can act as a restriction:

```
user=> (for [x (range 8) :when (even? x)] x)
(0 2 4 6)
```

Let's move on to look at Clojure implements laziness, especially as applied to sequences.

14.4.2 Lazy sequences

In Clojure laziness is mostly commonly seen when working with sequences, rather than lazy evaluation of a single value. For sequences, laziness means that rather than having a complete list of every value that's in a sequence, values can instead be obtained when they're required (such as by calling a function to generate them on demand).

In the Java Collections, such an idea would require something like a custom implementation of `List`, and there would be no convenient way to write it without large amounts of boilerplate code.

Using an implementation of `ISeq`, on the other hand, would allow us to write something like this:

```
public class SquareSeq implements ISeq {
    private final int current;

    private SquareSeq(int current) {
        this.current = current;
    }

    public static SquareSeq of(int start) {
        if (start < 0) {
            return new SquareSeq(-start);
        }
        return new SquareSeq(start);
    }

    @Override
    public Object first() {
        return Integer.valueOf(current * current);
    }

    @Override
    public ISeq rest() {
        return new SquareSeq(current + 1);
    }
}
```

There is no storage of values, and instead each new element of the sequence is generated on-demand. This allows us to model infinite sequences.

Or consider this example:

```

public class IntGeneratorSeq implements ISeq {
    private final int current;
    private final Function<Integer, Integer> generator;

    private IntGeneratorSeq(int seed,
                           Function<Integer, Integer> generator) {
        this.current = seed;
        this.generator = generator;
    }

    public static IntGeneratorSeq of(int seed,
                                    Function<Integer, Integer> generator) {
        return new IntGeneratorSeq(seed, generator);
    }

    @Override
    public Object first() {
        return generator.apply(current);
    }

    @Override
    public ISeq rest() {
        return new IntGeneratorSeq(generator.apply(current), generator);
    }
}

```

This example uses the result of applying the function to provide the seed of the next sequence. This is fine, provided that the generator function is pure, but of course nothing guarantees that in Java.

Let's move on and take a look at some powerful Clojure macros designed to help you create lazy seqs with only a small amount of effort.

Consider how you could represent a lazy, potentially infinite sequence. One obvious choice would be to use a function to generate items in the sequence. The function should do two things:

- Return the next item in a sequence
- Take a fixed, finite number of arguments

Mathematicians would say that such a function defines a recurrence relation, and the theory of such relations immediately suggests that recursion is an appropriate way to proceed.

Imagine you have a machine in which stack space and other constraints aren't present, and suppose that you can set up two threads of execution: one will prepare the infinite sequence, and the other will use it. Then you could use recursion to define the lazy seq in the generation thread with something like the following snippet of pseudocode:

```
(defn infinite-seq <vec-args>
  (let [new-val (seq-fn <vec-args>)]
    (cons new-val (infinite-seq <new-vec-args>)) ①)
  ))
```

- ① This doesn't actually work, because the recursive call to (`infinite-seq`) blows the stack up.

The solution is to add a construct that tells Clojure to optimize the recursion away, and only proceed as needed - the `(lazy-seq)` macro.

Let's look at a quick example that defines the lazy sequence $k, k+1, k+2, \dots$ for some number k .

Listing 14.1 Lazy sequence example

```
(defn next-big-n [n] (let [new-val (+ 1 n)]
  (lazy-seq
    (cons new-val (next-big-n new-val)))
  )))
(defn natural-k [k]
  (concat [k] (next-big-n k)))
1:57 user=> (take 10 (natural-k 3))
(3 4 5 6 7 8 9 10 11 12)
```

- ① lazy-seq marker
- ② Infinite recursion
- ③ concat constrains recursion

The key points are the form `(lazy-seq)`, which marks a point where an infinite recursion could occur, and the `(concat)` form, which handles it safely. You can then use the `(take)` form to pull the required number of elements from the lazy sequence.

Lazy sequences are an extremely powerful feature, and with practice you'll find them a very useful tool in your Clojure arsenal.

14.4.3 Currying in Clojure

Currying functions in Clojure has an additional complexity compared to other languages. This is caused by the fact that many Clojure forms are variadic, as we discussed in Chapter 10.

Variadic functions are a complication because they raise questions like: "Does the user mean to curry the 2-argument form or evaluate the 1-argument form?" - especially as Clojure uses eager evaluation of functions.

The solution is the `(partial)` form - which, by the way, is a genuine Clojure function, not a macro. Let's see it in action:

```
user=> (filter (partial = :b) [:a :b :c :b :d])
(:b :b)
```

The function `=` takes 1 or more arguments, and so `(= :b)` would eagerly evaluate to `true`, but the use of `(partial)` turns it into a curried function. Its use in a `(filter)` call causes it to be recognized as a function of one argument (effectively, the 1-argument overload) and then it is used to test each element in the following vector.

NOTE

`(partial)` by itself will only curry the first parameter of a form. If we wanted to curry some other parameter, we would need to combine `(partial)` with another form - one that permuted the argument list before function application.

Clojure is by far the most functional of the three languages we have looked at, and if the treatment here has wetted your appetite, then there is a great deal more that you can explore - what we have covered so far is only the beginning, but it does help to demonstrate that the JVM itself can be a good home for functional programming - it's more the Java language that encumbers programming in a functional style.

14.5 Summary

In this chapter, we have delved far deeper into functional programming than just the traditional filter-map-reduce paradigm of Java Streams. We have largely done so by moving outside of Java to other JVM languages.

We've met techniques such as:

- Closures
- Lazy evaluation
- Currying
- Immutability

It is, of course, possible to go even further than this - and there are two major schools of functional programming - the dynamically typed school as represented by Clojure (and languages like Erlang outside of the JVM) and the statically typed school - which includes Kotlin but which is perhaps better represented by Scala (and Haskell for non-JVM languages).

However, one of Java's major design virtues - backwards compatibility - can also be seen as a potential weakness. Java code that was compiled for version 1.0 (over 25 years ago) will still run without modification on modern JVMs.

However, this remarkable achievement does not come without a price tag. Java's APIs and even the design of the bytecode and VM have to live with design decisions that are difficult or impossible to change now and which are not especially friendly to FP.

In the next chapter, we'll take what we've learned here and see how it applies to concurrent programming, and see how FP and concurrency are combining in modern practice to produce some extremely powerful techniques.

Notes

1. stackoverflow.com/questions/62950667/is-there-any-need-to-switch-to-modules-when-migrating-to-java-9-java-11/629