O'REILLY®

Java 9 Modularity

PATTERNS AND PRACTICES FOR
DEVELOPING MAINTAINABLE APPLICATIONS

Sander Mak & Paul Bakker

Java 9 Modularity

Sander Mak and Paul Bakker

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Java 9 Modularity
by Sander Mak and Paul Bakker

Printed in the United States of America.

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

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

Editors: Nan Barber and Brian Foster
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2016-08-11: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491954096* for release details.

978-1-491-95409-6

[FILL IN]

Table of Contents

Preface Title

This Is an A-Head

Congratulations on starting your new project! We've added some skeleton files for you, to help you get started, but you can delete any or all of them, as you like. In the file called chapter.html, we've added some placeholder content showing you how to markup and use some basic book elements, like notes, sidebars, and figures.

CHAPTER 1
Modularity Matters

Have you ever scratched your head in bewilderment, asking yourself: "Why is this code here? How does it relate to the rest of this gigantic codebase? Where do I even begin?" Or, did your eyes glaze over after scanning the multitude of Java Archives (JARs) bundled with your application code? We certainly have.

The art of structuring large codebases is an undervalued one. This is neither a new problem, nor is it specific to Java. However, Java is one of the mainstream languages in which very large applications are built all the time. Often making heavy use of many libraries from the Java ecosystem. Under these circumstances systems can outgrow our capacity for understanding and efficient development. A lack of structure is dearly paid for in the long run, experience shows.

Modularity is one of the tools you can employ to manage and reduce this complexity. Java 9 introduces a new module system that makes modularization easier and more accessible. It builds on top of abstractions Java already has for modular development. In a sense, it promotes existing best practices on large-scale Java development to be part of the Java language.

The Java module system will have a profound impact on Java development. It represents a fundamental shift to modularity as a first-class citizen for the whole Java platform. Modularization is addressed from the ground up, with changes to the language, Java Virtual Machine (JVM) and standard libraries. While this represents a monumental effort, it's not as flashy as for example the addition of Streams and Lambda's in Java 8. There's another fundamental difference between a feature like Lambda's and the Java module system. A module system is concerned with the large-scale structure of whole applications. Turning an inner class into a lambda is a fairly small and localized change within a single class. Modularizing an application affects design, compilation, packaging, deployment and so on. Clearly, it's much more than just another language feature.

With every new Java release, it's tempting to dive right in and start using the new features. To make the most out of the module system, we should first take a step back and focus on what modularity is. And, more importantly, why we should care.

What is Modularity?

So far, we've touched upon the goal of modularity (managing and reducing complexity), but not what modularity entails. At its heart, modularization is the act of decomposing a system into self-contained modules. Modules are identifiable artifacts containing code, with metadata describing the module and its relation to other modules. Ideally, these artifacts are recognizable from compile-time all the way through run-time. An application then consists of several modules working together.

So, modules group related code, but there's more to it than just that. There are three core tenets modules must adhere to:

Strong encapsulation
: A module must be able to conceal part of its code from other modules. By doing so, a clear line is drawn between code that is publicly usable, and code that is deemed an internal implementation detail. This prevents accidental or unwanted coupling between modules: you simply cannot use what has been encapsulated. Consequently, encapsulated code may change freely without affecting users of the module.

Well-defined interfaces
: Encapsulation is fine, but if modules are to work together, not everything can be encapsulated. Code that is not encapsulated is, by definition, part of the public API of a module. Since other modules can use this public code, it must be managed with great care. A breaking change in non-encapsulated code can break other modules that depend on it. Therefore, modules should expose well-defined and stable interfaces to other modules.

Explicit dependencies
: Modules often need other modules to fulfill their obligations. Such dependencies must be part of the module definition, in order for modules to be self-contained. Explicit dependencies give rise to a *module graph*: nodes represent modules, and edges represent dependencies between modules. Having a module graph is important for both understanding an application and running it with all necessary modules. It provides the basis for a reliable configuration of modules.

With modules, flexibility, understandability and reusability all come together. Modules can be flexibly composed in different configurations, making use of the explicit dependencies to ensure a working configuration. Encapsulation ensures you never have to know implementation details and that you will never accidentally rely on them. To use a module, knowing its public API is enough. Also, a module exposing

well-defined interfaces while encapsulating its implementation details can readily be swapped with alternative implementations.

Modular applications have many advantages. Experienced developers know all too well what happens when codebases are non-modular. Endearing terms like "spaghetti architecture", "monolith" or "big ball of mud" do not even begin to cover the associated pain. Modularity is not a silver bullet, though. It is an architectural tool that can *prevent* these problems to a high degree when applied correctly.

That being said, the definition of modularity provided in this section is deliberately abstract. It possibly makes you think of component-based development (all the rage in the previous century), Service Oriented Architecture or the current micro-services hype. Indeed, these paradigms try to solve similar problems at various levels of abstraction.

What would it take to realize modules in Java? It's instructive to take a moment and think about how the core tenets of modularity are already present in Java as you know it (and where it is lacking).

Done? Then you're ready to proceed to the next section.

Before Java 9

Java is used for development of all sorts and sizes. Applications comprising millions of lines of code are no exception. Evidently, Java has done something right when it comes to building large-scale systems. Even before Java 9 arrived on the scene. Let's examine the three core tenets of modularity again in the light of Java before the arrival of the Java 9 module system.

Encapsulation of types can be achieved by using a combination of *packages* and *access modifiers* (like `private`, `protected` or `public`). By making a class `protected`, for example, you can prevent other classes from accessing it unless they reside in the same package. That does raise an interesting question: what if you do want to access that class from another package in your code, but still want to prevent others from using it? The only way to achieve this is by making the class public. Public means public to every other package in the system, thus voiding the encapsulation. Of course, you can hint that using such a class is not smart by putting it in an `.impl` or `.internal` package. But really, who looks at that? People use it anyway, just because they can.

In the well-defined interfaces department, Java has been doing great since its inception. You guessed it, we're talking about interfaces. Exposing a public interface, while hiding the implementation class behind a factory or through dependency injection is a tried and true method. As we will see throughout this book, interfaces play a central role in modular systems.

Explicit dependencies is where things are starting to fall apart. Yes, Java does have explicit import statements. Unfortunately, those imports are strictly a compile-time construct. Once you package your code into a JAR, there's no telling which other JARs contain the types your JAR needs to actually run. In fact, this problem is so bad, many external tools evolved alongside the Java language to solve this problem. "External Tooling to Manage Dependencies: Maven and OSGi" on page 10 provides more details.

External Tooling to Manage Dependencies: Maven and OSGi

Maven

One of the problems solved by the Maven build tool is compile-time dependency management. Dependencies between JARs are defined in an external POM (Project Object Model) file. Maven's great success is not the build tool per se, but the fact that it spawned a canonical repository called Maven Central. Virtually all Java libraries are published along with their POMs to Maven Central. Various other build tools like Gradle or Ant (with Ivy) use the same repository and metadata. They all automatically resolve (transitive) dependencies for you at compile-time.

OSGi

What Maven does at compile-time, OSGi does at run-time. OSGi requires imported packages to be listed as metadata in JARs, which are then called bundles. You must also explicitly define which packages are exported, that is, visible to other bundles. At application start, all bundles are checked: can every importing bundle be wired to an exporting bundle? A clever setup of custom classloaders ensures at run-time no types are loaded in a bundle besides what is allowed by the metadata. As with Maven, this does require the whole world to provide correct OSGi metadata in their JARs. However, where Maven has unequivocally succeeded with Maven Central and POMs, the proliferation of OSGi-capable JARs is less impressive.

Both Maven and OSGi are built on top of the JVM and Java language, which they do not control. Java 9 addresses these same problems in the core of the JVM and the language.

As it stands, Java offers solid constructs for creating large-scale modular applications. It's also clear there is definitely room for improvement.

JARs as Modules?

JAR files seem to be the closest we can get to modules pre-Java 9. They have a name, group related code and can offer well-defined public interfaces. Let's look at an example of a typical Java application running on top of the JVM to explore the notion of JARs as modules:

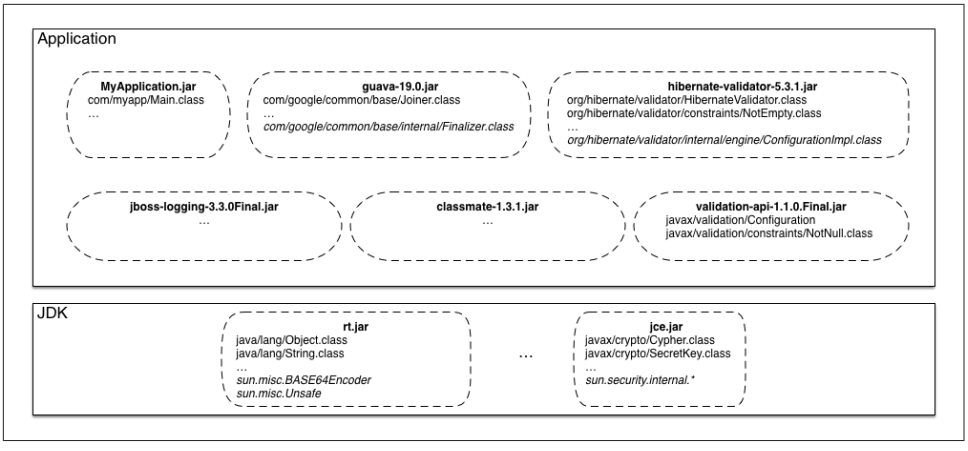

Figure 1-1. MyApplication is a typical Java application, packaged as JAR and using other libraries.

There's an application JAR `MyApplication.jar` containing custom application code. Two libraries are used by the application: Google Guava and Hibernate Validator. There are three additional JARs as well. Those are transitive dependencies of Hibernate Validator, possibly resolved for us by a build tool like Maven. MyApplication runs on top of a (pre-Java 9) JVM, which itself exposes Java platform classes through several bundled JARs. In the Java runtime, `rt.jar` (*runtime* library) is the most prominent platform JAR, containing all classes of the Java standard library.

When you look closely at Figure 1-1, you can see some of the JARs list classes in *italic*. These classes are supposed to be internal classes of the libraries. For example, `com.google.common.base.internal.Finalizer` is used in Guava itself, but is not part of the official API. It's a public class, since other Guava packages make use of `Finalizer`. Unfortunately this also means there's no impediment for `com.myapp.Main` to use classes like `Finalizer`. In other words, there's no encapsulation.

The same holds for internal classes from the Java platform itself. Packages like `sun.misc` have always been accessible to application code, even though documentation sternly warns they are unsupported APIs. Despite this warning, utility classes such as `sun.misc.BASE64Encoder` are used in application code all the time. Technically, that code *may* break with any update of the JDK, since they are internal implementation classes. Lack of encapsulation essentially forced those classes to be considered semi-public API anyway, since Java values backwards compatibility highly. An unfortunate situation, arising from the lack of true encapsulation.

What about explicit dependencies? As we've already learned, there is no dependency information anymore when looking strictly at JARs. You run MyApplication as follows:

```
java -classpath lib/guava-19.0.jar:\
                lib/hibernate-validator-5.3.1.jar:\
                lib/jboss-logging-3.3.0Final.jar:\
                lib/classmate-1.3.1.jar:\
                lib/validation-api-1.1.0.Final.jar \
     -jar MyApplication.jar
```

Setting up the correct classpath is up to the user. And, without explicit dependency information, it is not for the faint of heart.

Classpath Hell

The classpath is used by the Java runtime to locate classes. In our example we run `Main`, and all classes that are directly or indirectly referenced from this class need to be loaded at some point. You can view the classpath as a list of all classes that *may* be loaded at runtime. While there is more to it behind the scenes, this view suffices to understand the issues with the classpath.

A condensed view of the resulting classpath for MyApplication looks like this:

```
java.lang.Object
java.lang.String
...
sun.misc.BASE64Encoder
sun.misc.Unsafe
...
javax.crypto.Cypher
javax.crypto.SecretKey
...
com.myapp.Main
...
com.google.common.base.Joiner
...
com.google.common.base.internal.Joiner
org.hibernate.validator.HibernateValidator
org.hibernate.validator.constraints.NotEmpty
...
org.hibernate.validator.internal.engine.ConfigurationImpl
...
javax.validation.Configuration
javax.validation.constraints.NotNull
```

There's no notion of JARs or logical grouping anymore. All classes are sequenced into a flat list, in the order defined by the `-classpath` argument. When the JVM loads a class, it reads the classpath in fixed order to find the right one. As soon as the class is found, the search ends and the class is loaded.

What if a class cannot be found on the classpath? Then you will get a run-time exception. Because classes are loaded lazily, this could potentially be triggered when some unlucky user clicks a button in your application for the first time. The JVM cannot efficiently verify the completeness of the classpath upon starting. There is no way to tell in advance whether the classpath is complete, or if you should add another jar. Obviously that's not good.

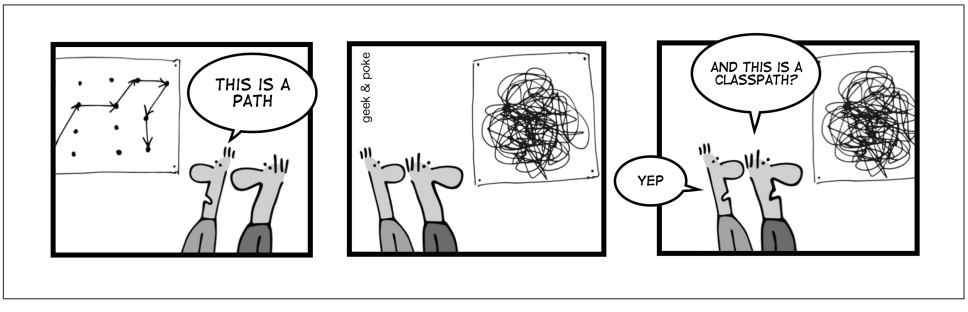

Figure 1-2. Graph Theory for Geeks by Oliver Widder, CC-BY

More insidious problems arise when duplicate classes are on the classpath. Let's say you try to circumvent the manual setup of the classpath. Instead, you let Maven construct the right set of JARs to put on the classpath, based on the explicit dependency information in POMs. Since Maven resolves dependencies transitively, it's not uncommon for two version of the same library (say Guava 19 and Guava 18) to end up in this set, through no fault of your own. Now both library JARs are flattened into the classpath, in some undefined order. Whichever version of the library classes come first, is loaded. However, some other classes may expect a class from the (possibly incompatible) other version. Again, this leads to run-time exceptions. In general, whenever the classpath contains two classes with same (fully qualified) name, even if they are completely unrelated, only one "wins".

It now becomes clear why the term "Classpath Hell" (also known as "JAR Hell") is so infamous in the Java world. Some people have perfected the art of tuning a classpath through trial-and-error. A rather sad occupation when you think about it. The fragile classpath remains a leading cause of problems and frustration. If only there were more information available about the relation between JARs at run-time. It's like there's a dependency graph hiding in the classpath that is just waiting to come out and be exploited. Enter Java 9 modules!

Java 9 Modules

By now, you have a solid understanding of Java's current strengths and limitations when it comes to modularity. With Java 9 we get a new ally in the quest for well-

structured applications: the Java module system. While designing the Java Platform Module System to overcome current limitations, two main goals were defined:

- Modularize the JDK itself.
- Offer a module system for applications to use.

These goals are closely related. Modularizing the JDK is done using the same module system that we, as application developers, can use in Java 9.

The module system introduces a native concept of modules into the Java language and runtime. Modules can either export or strongly encapsulate packages. Furthermore, they express dependencies on other modules explicitly. As you can see, all three tenets of modularity are addressed by the Java module system.

Let's revisit the MyApplication example, now based on the Java 9 module system:

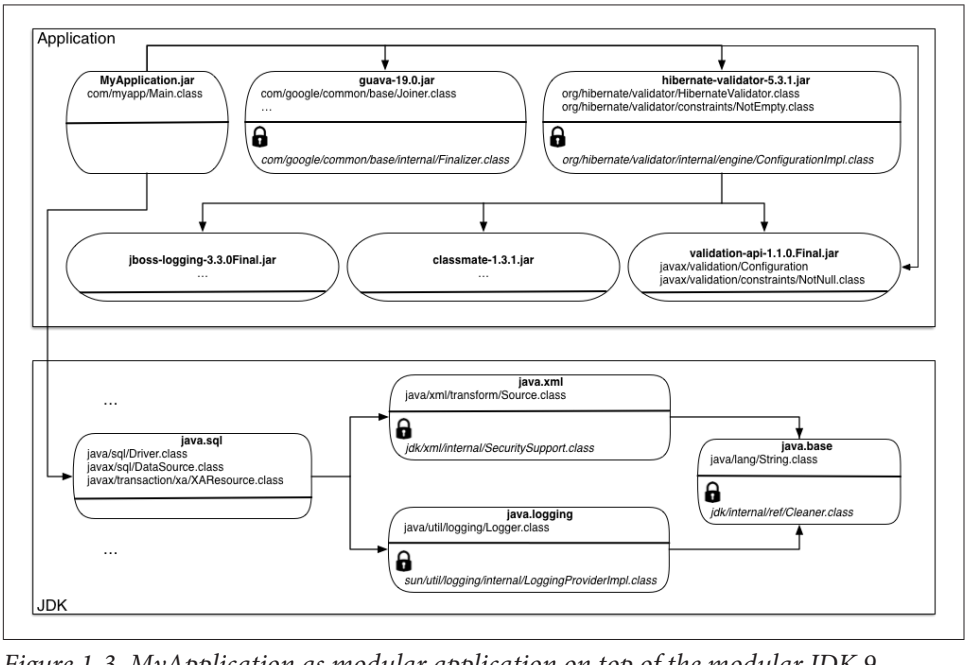

Figure 1-3. MyApplication as modular application on top of the modular JDK 9.

Each JAR becomes a module, containing explicit references to other modules. The fact that hibernate-validator uses jboss-logging, classmate and validation-api is part of its *module descriptor*. A module has a publicly accessible part (on the top) and an encapsulated part (on the bottom, indicated with the padlock). That's why MyApplication can no longer use Guava's Finalizer class. Through this diagram, we discover that MyApplication uses validation-api as well, to annotate some of its classes. What's

more, MyApplication has an explicit dependency on a module in the JDK called `java.sql`.

`???`. All that could be said there is that MyApplication uses classes from `rt.jar`, like all Java applications. And, that it runs with a bunch of JARs on the (possibly incorrect) classpath.

That's just the application layer. It's modules all the way down. At the JDK layer, there are modules as well. Like the modules in the application layer, they have explicit dependencies and expose some packages while concealing others. The most essential *platform module* in the modular JDK is `java.base`. It exposes packages like `java.lang` and `java.util` which no other module can do without. Because you cannot avoid using these types from these packages, every module requires `java.base` implicitly. If the application modules require any functionality from platform modules other than what's in `java.base`, these dependencies should be explicit as well, as is the case with MyApplication's dependency on `java.sql`.

Finally there's a way to express dependencies between separate parts of the code at a higher level of granularity in the Java language. Now imagine the advantages of having all this information available at compile-time and run-time. Accidental dependencies on code from other non-referenced modules can be prevented. The toolchain knows which modules are necessary for running a module by inspecting its (transitive) dependencies and optimizations can be applied using this knowledge.

Strong encapsulation, well-defined interfaces and explicit dependencies are now part of the Java platform. In short, these are the most important benefits of the Java Platform Module System:

Reliable configuration
: The module system checks whether a given combination of modules satisfies all dependencies before compiling or running code. This leads to less run-time errors.

Strong encapsulation
: Modules explicitly choose what to expose to other modules. Accidental dependencies upon internal implementation details are prevented.

Scalable development
: Explicit boundaries enable teams to work in parallel while still creating maintainable codebases. Only explicitly exported public types are shared, creating boundaries that are automatically enforced by the module system.

Security
: Strong encapsulation is enforced at the deepest layers inside the JVM. This limits the attack surface of the Java runtime. Gaining reflective access to sensitive internal classes is not possible anymore.

Optimization
: Because the module system knows which modules belong together, no other code needs to be considered during JVM startup. It also opens up the possibility to create a minimal configuration of modules for distribution. Furthermore, whole-program optimizations can be applied to such a set of modules. Before modules, this was impossible because explicit dependency information was not available and a class could reference any other class from the classpath.

In the next chapter, we explore how modules are defined and what concepts govern their interactions. We do this by looking at modules in the JDK itself. There are many more platform modules than shown in Figure 1-3.

It's is a great way to get to know the module system concepts, while at the same time familiarizing yourself with the modules in the JDK. These are, after all, the modules you'll be using first and foremost in your modular Java 9 applications. After that, you're ready to start writing your own modules in Chapter 3.

CHAPTER 2
Modules and the Modular JDK

Java is over twenty years old. As a language it's still very popular, proving that Java has held up very well. The platform's long evolution becomes especially apparent when looking at the standard libraries. Prior to the Java module system, the runtime library of the JDK consisted of a hefty `rt.jar` (as shown previously in Figure 1-1), weighing in at more than 60 Megabytes. It contains most of the runtime classes for the Java platform: the ultimate monolith of the Java platform. In order to regain a flexible and future-proof platform, the JDK team set out to modularize the JDK. An ambitious goal, given the size and structure of the JDK. Over the course of the past twenty years, many APIs have been added. Virtually none have been removed.

Take CORBA. Once considered the future of enterprise computing, now a mostly forgotten technology (to those who are still using it: we feel for you). The classes supporting CORBA in the JDK are still present in `rt.jar` to this day. Each and every distribution of Java, regardless of the applications it runs, includes those CORBA classes. No matter if you use CORBA or not; it's just there. Carrying this legacy in the JDK results in unnecessary use of disk space, memory and CPU time. In the context of resource-constrained devices, or when creating small containers for the cloud, these resources are in short supply. Not to mention the cognitive overhead of obsolete classes showing up in IDE auto-completions and documentation during development.

Simply removing these technologies from the JDK isn't a viable option though. Backwards compatibility is one of the most important guiding principles for Java. Removal of APIs would break a long streak of backwards compatibility. Although it may only affect a small percentage of users, there are still plenty of people using technologies like CORBA. In a modular JDK, people who aren't using CORBA can simply choose to ignore the module containing CORBA.

Alternatively, an aggressive deprecation schedule for truly obsolete technologies could work. Still, it would take several major releases before the JDK really sheds the excess weight. Also, deciding what technology is truly obsolete would be at the discretion of the JDK team, which is a difficult position to be in.

In the specific case of CORBA, the module is marked as deprecated, meaning it will likely be removed in a subsequent major Java release.

But the desire to break up the monolithic JDK is not just about removing obsolete technology. There also is a vast array of technologies that are useful to certain types of applications, while useless for others. JavaFX is the latest user-interface technology in Java, after AWT and Swing. This is certainly not something to be removed, but clearly it's not required in every application either. Web-applications, for example, use none of the GUI toolkits in Java. Yet there is no way to deploy and run them without all three GUI toolkits being carried along.

Aside from convenience and waste, there's also the security perspective. Java has experienced a considerable number of security exploits in the past years. Many of these exploits share a common trait: somehow, attackers gain access to sensitive classes inside the JDK to bypass the JVM's security sandbox. By simply decreasing the number of classes in the runtime, the attack surface decreases. Having tons of unused classes around in your application runtime only for them to be exploited later is an unfortunate tradeoff. With a modular JDK, again, you can choose to include just those platform modules you need. Strongly encapsulating dangerous internal classes within platform modules that you do need is another improvement from a security standpoint.

By now, it's abundantly clear that a modular approach for the JDK itself is sorely needed.

The Modular JDK

The first step towards a more modular JDK was taken in Java 8 with the introduction of *compact profiles*. A profile defines a subset of packages from the standard library available to applications targeting that profile. Three profiles are defined, imaginatively called compact1, compact2 and compact3. Each of these profiles is a superset of the previous, adding more packages that can be used. The Java compiler and runtime were updated with knowledge of these pre-defined profiles. Java SE Embedded 8 offers low footprint runtimes matching the compact profiles.

If your application fits one of the profiles, this is a good way to target a smaller runtime. But, if you even require so much as a single class outside of the pre-defined profiles, you're out of luck. In that sense, compact profiles are far from flexible. They also don't address encapsulation. As an intermediate solution, compact profiles fulfilled their purpose. Ultimately, a more flexible approach is needed.

We already saw a glimpse of how JDK 9 is split up into modules in Figure 1-3. The JDK now consists of about 90 platform modules, instead of a monolithic library. Every module constitutes a well-defined piece of functionality of the JDK, ranging from logging to XML support. All modules explicitly define their dependencies on other modules.

A subset of these platform modules and their dependencies is shown in Figure 2-1. Every arrow indicates a uni-directional dependency between modules (we'll get to the difference between solid and dashed arrows later). For example, `java.xml` depends on `java.base`. As stated in "Java 9 Modules" on page 13, every module implicitly depends on `java.base`. In Figure 2-1 this implicit dependency is only shown when `java.base` is the sole dependency for a given module, as is the case with `java.xml`

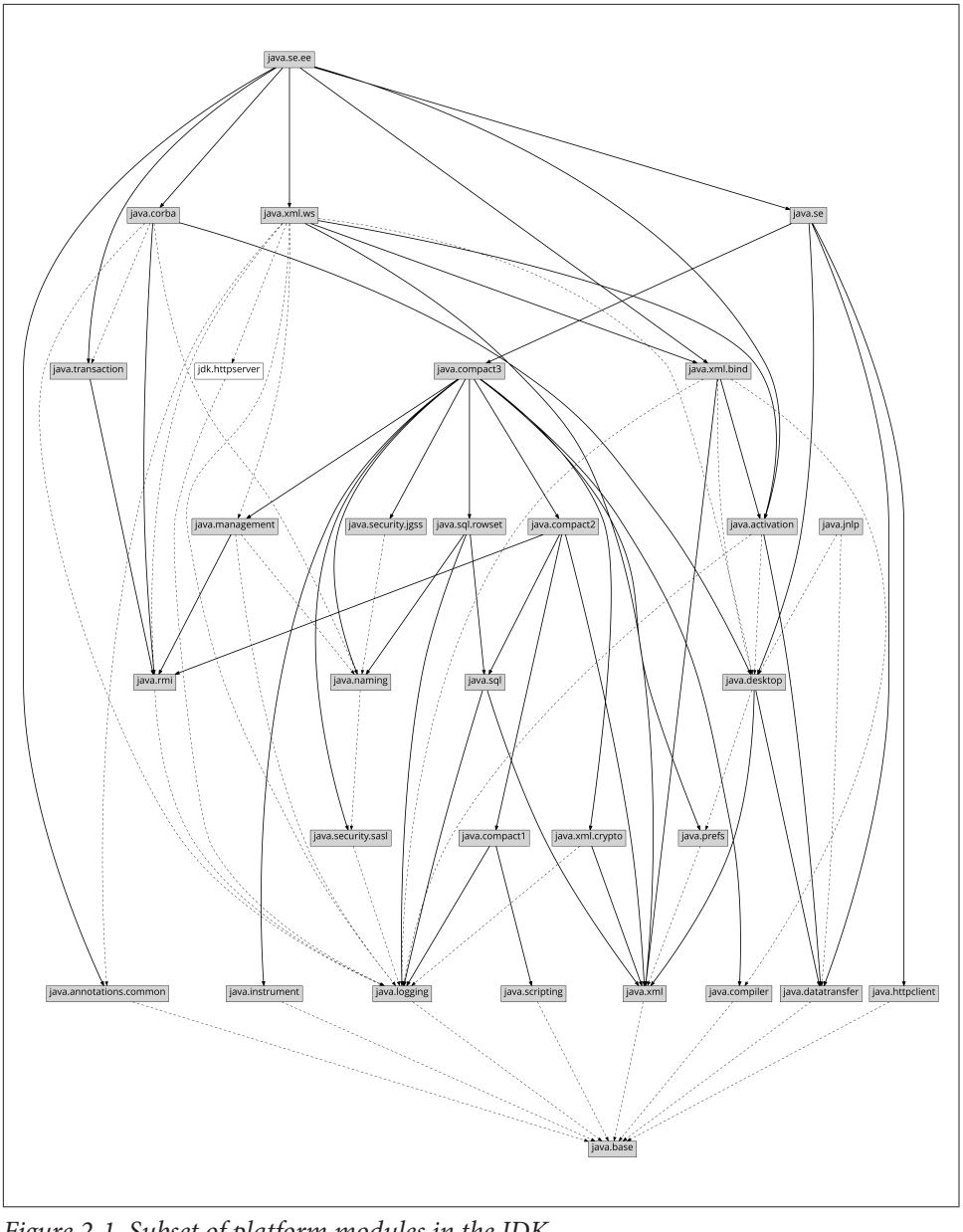

Figure 2-1. Subset of platform modules in the JDK.

Even though the dependency graph may look a little bit overwhelming, we can glean a lot of information from it. Just by looking at the graph, you can get a decent overview of what the Java standard libraries offer and how the functionalities are related. For example, `java.logging` has many *incoming dependencies*, meaning it used by

many other platform modules. That makes sense for a central functionality such as logging. Module `java.xml.ws` (containing the JAX-WS APIs for SOAP WebServices) has many *outgoing dependencies*, including an unexpected one on `java.desktop`.

Another thing to note is how all dependencies point downward. There are no cycles in this graph. That's not by accident: the Java module system does not allow circular dependencies between modules. Having an overview of a large codebase like this, based on explicit module information is invaluable.

Circular dependencies are generally an indication of bad design. In ??? we discuss how you can identify and resolve circular dependencies in your codebase.

All modules in Figure 2-1, except `jdk.httpserver`, are part of the Java SE specification. They share the `java.*` prefix for module names. Modules like `jdk.httpserver` contain *implementations* of tools and APIs. Where such implementations live is not mandated by the Java SE specification, but of course such modules are essential to a fully functioning Java platform. There are many more modules in the JDK, most of them in the `jdk.*` namespace.

You can get the full list of platform modules by running `java --list-modules`.

Two important modules can be found at the top of Figure 2-1: `java.se` and `java.se.ee`. These are so-called *aggregator modules* and they serve to logically group several other modules. Aggregator modules roughly corresponding to the previously discussed compact profiles have been defined as well: `java.compact1`, `java.compact2` and `java.compact3`.

Decomposing the JDK into modules has been a tremendous amount of work. Splitting up an entangled, organically grown codebase containing tens of thousands of classes into well-defined modules with clear boundaries, while retaining backwards compatibility, takes time. This is one of the reasons it took a long time to get a module system into Java. With over 20 years of legacy accumulated, many dubious dependencies had to be untangled. Going forward, this effort will definitely pay off in terms of development speed and increased flexibility for the JDK.

Module Descriptors

Now that we have a high level overview of the JDK module structure, let's explore how it works. What is a module, and how is it defined? A module has a name, it groups related code and possibly other resources, and is described by a module descriptor. The module descriptor lives in a file called *module-info.java*. Here's an example of the module descriptor for the `java.prefs` platform module:

Example 2-1. module-info.java

```
module java.prefs {
    requires java.xml; ❶

    exports java.util.prefs; ❷
}
```

❶ The `requires` keywords indicates a dependency, in this case on `java.xml`.

❷ A single package from the `java.prefs` module is exported to other modules.

Modules live in a global namespace, therefore module names must be unique. As with package names, you can use conventions like reverse DNS notation (e.g. `com.mycompany.project.somemodule`) to ensure uniqueness for your own modules. A module descriptor always starts with the `module` keyword, followed by the name of the module. Then, the body of module-info.java describes other characteristics of the module, if any.

Let's move on to the body of the module descriptor for `java.prefs`. Code in `java.prefs` uses code from `java.xml` to load preferences from XML files. This dependency must be expressed in the module descriptor. Without this dependency declaration, the `java.prefs` module would not compile (or run). Such a dependency is declared with the `requires` keyword followed by a module name, in this case `java.xml`. The implicit dependency on `java.base` may be added to a module descriptor. Doing so adds no value, similar to how you can (but generally don't) add "`import java.lang.String`" to code using Strings.

A module descriptor can also contain exports statements. Strong encapsulation is the default for modules. Only when a package is explicitly exported, like `java.util.prefs` in this example, can it be accessed from other modules. Packages inside a module that are not exported, are inaccessible from other modules by default. Other modules cannot refer to types in encapsulated packages, even if they have dependency on the module. When you look at Figure 2-1 you see that `java.desktop` has a dependency on `java.prefs`. That means `java.desktop` is only able to access types in package `java.util.prefs` of the `java.prefs` module.

Readability

An important new concept when reasoning about dependencies between modules is *readability*. Reading another module means you can access types from its exported packages. You set up readability relations between modules through requires clauses in the module descriptor. By definition, every module reads itself. A module that requires another module *reads* the other module.

Let's explore the effects of readability by revisiting the `java.prefs` module. In this JDK module, the following class imports and uses classes from the `java.xml` module:

Example 2-2. Small excerpt from the class `java.util.prefs.XmlSupport`.

```
import org.w3c.dom.Document;
// ...

class XmlSupport {

  static void importPreferences(InputStream is)
      throws IOException, InvalidPreferencesFormatException
  {
      try {
          Document doc = loadPrefsDoc(is);
          // ...
        }
  }

  // ...
}
```

Here, `org.w3c.dom.Document` (amongst other classes) is imported. It comes from the `java.xml` module. Because the `java.prefs` module descriptor contains `requires java.xml`, this compiles without issue. Had the author of the `java.prefs` module left out the requires clause, the Java compiler would report an error. Using code from `java.xml` in module `java.prefs` is a deliberate and explicitly recorded choice.

Readability is not transitive by default. We can illustrate this by looking at the incoming and outgoing read edges of `java.prefs`:

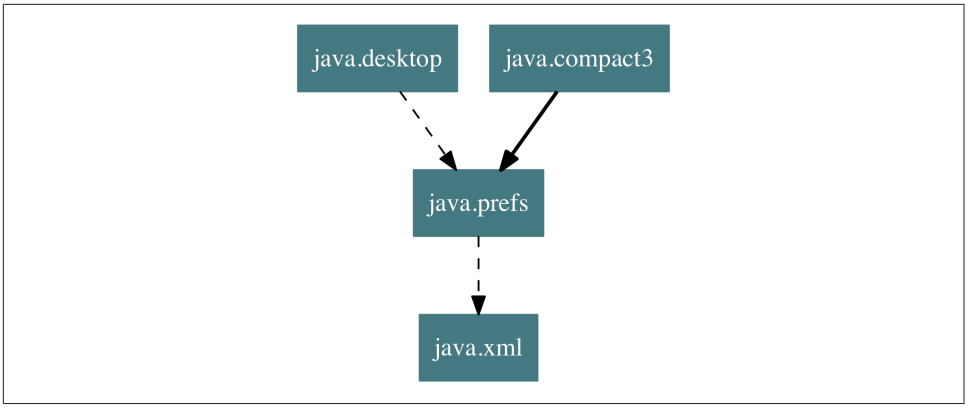

Figure 2-2. Readability is not transitive: `java.desktop` does not read `java.xml` through `java.prefs`.

Here, `java.desktop` reads `java.prefs` (amongst other modules, left out for brevity). We've already established this means `java.desktop` can access types from the `java.util.prefs` package. However, `java.desktop` cannot access types from `java.xml` through its dependency on `java.prefs`. It so happens that `java.desktop` *does use* types from `java.xml`. That's why `java.desktop` has its own `requires java.xml` clause in its module descriptor. In Figure 2-1 this dependency is also shown.

Implied readability

Sometimes, you do want read relations to be transitive. For example, when a type in an exported package of module M1 refers to a type from another module M2. In that case, M1 cannot be used without reading M2 as well.

A good example of this phenomenon can be found in the JDK's `java.sql` module. It contains two interfaces defining method signatures whose result types come from other modules:

Example 2-3. Driver interface (partially shown), allowing a `Logger` from the `java.logging` module to be retrieved.

```
package java.sql;

import java.util.logging.Logger;

public interface Driver {
  public Logger getParentLogger();
}
```

Example 2-4. XMLSQL interface (partially shown), with `Source` from module `java.xml` representing XML coming back from the database.

```
package java.sql;

import javax.xml.transform.Source;

public interface SQLXML {
  <T extends Source> T getSource(Class<T> sourceClass);
}
```

If you add a dependency on `java.sql` to your module descriptor, you can program to these interfaces, since they are in exported packages. But, whenever you call `getParentLogger()` or `getSource()`, you get back values of a type not exported by `java.sql`. In the first case you get a `java.util.logging.Logger` from `java.logging` and in the second case you get a `java.xml.transform.Source` from `java.xml`. In order to do anything useful with these return values (assign to a local variable, call methods on them), you need to read those other modules as well.

Of course, you can manually add dependencies on respectively `java.logging` or `java.xml` to your own module descriptor. But that's hardly satisfying, especially since the `java.sql` author already knew those interfaces are unusable without readability on those other modules. Implied readability allows module authors to express this transitive readability relation in module descriptors.

For `java.sql`, it looks like this:

```
module java.sql {
    requires public java.logging;
    requires public java.xml;

    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
}
```

The requires keyword is now followed by the `public` modifier, slightly changing the semantics. A normal requires allows a module to access types in exported packages from the required module only. Requires public means the same and then some more. In addition, any module requiring `java.sql` will now automatically be requiring `java.logging` and `java.xml`. That means you get access to the exported packages of those modules as well by virtue of these implied readability relations. With requires public, module authors can setup additional readability relations for users of the module.

From the consumer side, this makes it easier to use `java.sql`. When you require `java.sql`, you not only get access to the exported packages `java.sql`, `javax.sql` and `javax.transaction.xa` (which are all exported by `java.sql` directly), but also to all

packages exported by modules `java.logging` and `java.xml`. It's as if `java.sql` re-exports those packages for you, courtesy of the implied readability relations it sets up with requires public. For an application module `app` using `java.sql`, this module definition suffices:

```
module app {
  requires java.sql;
}
```

With this module descriptor, the following readability edges are in effect:

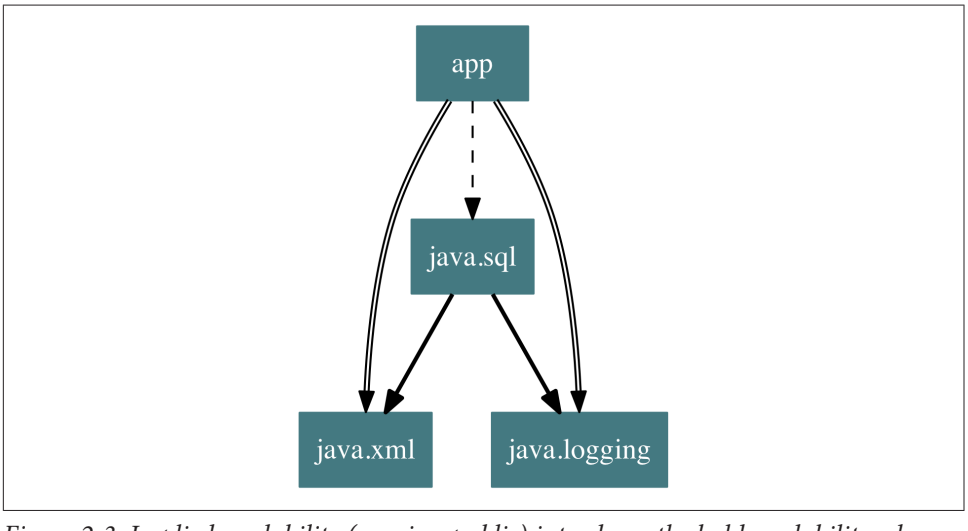

Figure 2-3. Implied readability (requires public) introduces the bold readability edges. Readability edges to `java.base` have been omitted for clarity.

Implied readability on `java.xml` and `java.logging` (the double-lined edges in Figure 2-3) is granted to `app` because `java.sql` uses `requires public` (bold edges in Figure 2-3) for those modules. Since `app` does not export anything and only uses `java.sql` for its encapsulated implementation, a normal requires clause is enough. In ??? we'll discuss how and when implied readability is important for your own modules in more detail.

Now's a good time to take another look at Figure 2-1. All bold edges in that graph are `requires public` dependencies, too. This highlights another use-case for implied readability: it can be used to aggregate several modules into a single new module. Take for example `java.se`. It's a module that doesn't contain any code and consists of just a module descriptor. In this module descriptor, a `requires public` clause is listed for each module that is part of the Java SE specification. When you require `java.se` in a module, you get access to all exported APIs of every module aggregated by `java.se`:

```
module java.se {
    requires public java.compact3;
    requires public java.datatransfer;
    requires public java.desktop;
    requires public java.httpclient;
}
```

Implied readability itself is transitive as well. That's why `java.se` can build on top of `java.compact3`, which in turn builds upon `java.compact2` and so on. In the end, `java.se` provides implied readability on a huge number of modules reachable through these transitive `requires public` dependencies. Figure 2-1 shows that `java.se.ee` provides yet another superset on top of `java.ee`. It adds several modules containing parts of the Java Enterprise Edition specification.

Use the aggregator modules available in the JDK with care. When your own modules would `require java.se` you're almost back to square one. Almost all packages available in the JDK are indiscriminately available to your module this way. It pays to be more precise in your dependencies.

In ??? we'll explore how the aggregator module pattern helps in modular library design.

Accessibility

Readability relations are about which modules read other modules. However, if you read a module this doesn't mean you can *access* everything from its exported packages. Normal Java accessibility rules are still in play after readability has been established.

Java has had accessibility rules built into the language since the beginning. Table 2-1 provides a refresher on the existing access modifiers and their impact.

Table 2-1. Access modifiers and their associated scopes.

Access modifier	Class	Package	Subclass	Unrestricted
public	✓	✓	✓	✓
protected	✓	✓	✓	
- (default)	✓	✓		
private	✓			

Accessibility is enforced at compile-time and run-time. Combining accessibility and readability provides the strong encapsulation guarantees we so desire in a module system. The question whether you can access a type from module M2 in module M1 becomes twofold:

1. Does M1 read M2?
2. If yes; is the type accessible in the package exported by M2?

Only public types in exported packages are accessible in other modules (ignoring situations where reflection APIs break through accessibility limitations). If a type is in an exported package but not public, traditional accessibility rules block its use. If it is public but not exported, the module system's readability rules block its use. Violations at compile-time result in compiler errors, whereas violations at run-time result in IllegalAccessErrors.

Is Public still Public?

No types from a non-exported package can be used from other modules. Even if the types inside that package are public. This is a fundamental change to the accessibility rules of the Java language.

Until Java 9, things were quite straightforward. If you had a public class or interface, it could be used by every other class. As of Java 9, `public` means public only to all other packages inside that module. Only when the package containing the public type is exported, can it be used by other modules. This is what strong encapsulation is all about. It forces developers to carefully design a package structure where types meant for external consumption are clearly separated from implementation details.

Before modules, the only way to strongly encapsulate implementation classes was to keep them all in a single package and mark them package-private. Since this leads to unwieldy packages, in practice classes were just made public for access across different packages. With modules, you can structure packages any way you like and only export those which really must be accessible to the consumers of the module. Exported packages form the API of a module, if you will.

Qualified Exports

In some cases you only want to expose a package to certain other modules. You can do this by using *qualified exports* in the module descriptor. An example of a qualified export can be found in the `java.xml` module:

```
module java.xml {
  ...
  exports com.sun.xml.internal.stream.writers to java.xml.ws
```

```
    ...
}
```

Here we see a platform module sharing useful internals with another platform module. The exported package is accessible only by the modules specified after `to`. Multiple module names, separated by comma, can be provided as target for a qualified export. Any module not mentioned in this `to` clause cannot access types in this package, even when they read the module.

The fact that qualified exports exist doesn't unequivocally mean you should use them. In general, avoid using qualified exports between modules in an application. Using them creates an intimate bond between the exporting module and its allowable consumers. From a modularity perspective, this is undesirable. One of the great things about modules is that you effectively decouple producers from consumers of APIs. Qualified exports break this property.

For modularizing the JDK, however, qualified exports have been indispensable. Many platform modules encapsulate part of their code, expose some internal APIs through qualified exports to select other platform modules, and use the normal export mechanism for public APIs used in applications. By using qualified exports, platform modules could be made more fine-grained without duplicating code.

Module Resolution and the Module Path

Having explicit dependencies between modules is not just useful to generate pretty diagrams. The Java compiler and runtime use module descriptors to resolve the right modules when compiling and running modules. Modules are resolved from the *module path*, as opposed to the classpath. Whereas the classpath is a flat list of types, the module path only contains modules. As we've learned, these modules carry explicit information on what packages they export, making the module path efficiently indexable. The Java runtime and compiler know exactly which module to load from the module path when looking for types from a given package. Previously, a scan through the whole classpath was the only way to locate an arbitrary type.

When you want to run an application packaged as module, you need all of its dependencies as well. Module resolution is the process of computing a minimal required set of modules given a dependency graph and a *root module* chosen from that graph. Every module reachable from the root module ends up in the set of *resolved modules*. Mathematically speaking, this amounts to computing the *transitive closure* of the dependency graph. As intimidating as it may sound, the process is quite intuitive:

1. Start with a single root module and add it to the resolved set.
2. Add each required module (`requires` or `requires public` in module-info.java) to the resolved set.

3. Repeat step 2 for each new module added to the resolved set in step 2.

This process is guaranteed to terminate because the dependency graph is acyclic by construction. If you want resolved modules for multiple root modules, just apply the algorithm to each root module, then take the union of the resulting sets.

Let's try this out with an example. We have an application module `app` that will be the root module in the resolution process. It only uses `java.sql` from the modular JDK:

```
module app {
    requires java.sql;
}
```

Now, we run through the steps of module resolution. We omit `java.base` when considering the dependencies of modules and assume it always is part of the resolved modules. You can follow along by looking at the edges in Figure 2-1.

1. Add `app` to the resolved set, observe that it requires `java.sql`.
2. Add `java.sql` to the resolved set, observe that it requires `java.xml` and `java.logging`.
3. Add `java.xml` to the resolved set, observe that it requires nothing else.
4. Add `java.logging` to the resolved set, observe that it requires nothing else.
5. No new modules have been added, resolution is complete.

The result of this resolution process is a set containing `app`, `java.sql`, `java.xml`, `java.logging` and `java.base`. When running `app`, the modules are resolved in this way and the module system loads the modules from the module path.

Some additional checks are performed during this process. For example, two modules with the same name lead to an error at start-up (rather than at run-time during inevitable class loading failures). Another check is for uniqueness of exported packages. Only one module on the module path may expose a given package. ??? discusses problems with multiple modules exporting the same packages.

The module resolution process and additional checks ensure the application runs in a reliable environment and is less likely to fail at run-time. In Chapter 3 we'll learn how to construct a module path when compiling and running our own modules.

Using the Modular JDK Without Modules

You've learned about many new concepts the module system introduces. At this point, you may be wondering how this all affects existing code, which obviously is not modularized yet. Do you really need to convert your code to modules to start using Java 9? Fortunately, no. Java 9 can be used as-is, without moving your code into

modules. The module systems is completely opt-in and the classpath is still alive and kicking.

Still, the JDK itself does consist of modules. How are these two worlds reconciled? Say you have the following piece of code:

```
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.LogRecord;


public class NotInModule {

  public static void main(String... args) {
    Logger logger = Logger.getGlobal();
    LogRecord message =
    new LogRecord(Level.INFO, "This still works!");
    logger.log(message);
  }

}
```

It's just a class, not put into any module. The code clearly uses types from the `java.logging` module in the JDK. However, there is no module descriptor to express this dependency. Still, when you compile this code without a module descriptor, put it on the classpath and run it, it will just work. How can this be? Code compiled and loaded outside of a module ends up in the *unnamed module*. In contrast, all modules we've seen so far are *named modules*, defining their name in module-info.java. The unnamed module is special: it reads all other modules, including the `java.logging` module in this case.

Through the unnamed module, code that is not yet modularized continues to run on JDK 9. Using the unnamed module happens automatically when you put your code on the classpath. That also means you are still responsible for constructing a correct classpath yourself. Almost all guarantees and benefits the module system offers are voided when working through the unnamed module.

There are two more things to be aware of, when using the classpath in Java 9. First, the unnamed module reads all other modules automatically, but can only access exported packages. As a result, types in modules that are not in an exported package are not accessible from the unnamed module. In JDK 9, many JDK internal implementation classes have been concealed in platform modules. Code that imports and uses those undocumented APIs fails to compile with Java 9 on its default settings. Here's an example of code using APIs that have been encapsulated in Java 9:

```
import sun.security.x509.X500Name;

public class ConcealedPlatformClass {
```

```
    public static void main(String... args) throws Exception {
      X500Name name = new X500Name("CN=user");
      System.out.println(name);
    }

}
```

When compiling this code with JDK 8, it does warn that `X500Name` is an internal proprietary API and should not be used. Besides these warnings, the code compiles and runs fine on JDK 8. However, when compiling the same code on JDK 9 you will encounter a compilation error. That's because the package containing `X500Name` is not exported from the `java.base` module. Hence, it is inaccessible from the unnamed module as well. When you compile this code on JDK 8, and try to run the resulting class file on JDK 9, it will blow up at run-time with an `IllegalAccessException`. Strong encapsulation in action!

In case you're wondering: the class you should be using instead of `X500Name` is `javax.security.auth.x500.X500Principal`. The corresponding package is exported by `java.base`, and is therefore accessible from the unnamed module.

The second thing to be aware of when compiling code in the unnamed module, is that `java.se` is taken as root module. You can access types from any module reachable by `java.se` and it will work. Conversely, this means modules under `java.se.ee` but not under `java.se` (such as `java.corba` and `java.xml.ws`) are not accessible. This choice was made because oftentimes people would put those types on the classpath themselves. In that case, your code continues to compile. If your code relies on those types coming from the JDK, it will fail to compile without additional action. These caveats, and their workarounds are discussed in more detail in the context of migration in ???.

In the next chapter, we'll take the module concepts discussed so far and use them to build our own modules.

CHAPTER 3

Working with Modules

In this chapter, we take the first steps towards modular development using Java 9. Instead of looking at existing modules in the JDK, it's time to get your hands dirty by writing your first module. To start things off easily, we turn our attention to the simplest possible module. Let's call it our *Modular Hello World*. Armed with this experience, we are then ready to take on a more ambitious example with multiple modules. At that point, we'll introduce the running example to be used throughout this book, called EasyText. It's designed to gradually grow with us, as we learn more about the module system.

Your First Module

We've seen examples of module descriptors in the previous chapter. A module is generally more than just a descriptor, though. The Modular Hello World therefore transcends the level of a single source file: we need to examine it in context. We start out by compiling, packaging and running a single module to get acquainted with the new tooling options for modules.

Anatomy of a Module

Our goal for this first example is to compile the following class into a module and run it. We start out with a single class in a package leading to a single module. Modules may only contain types that are part of packages, so a package definition is required.

```
package com.javamodularity.helloworld;

public class HelloWorld {

   public static void main(String... args) {
      System.out.println("Hello Modular World!");
   }
```

```
}
```

The layout of the sources on the filesystem looks as follows:

```
src/
  helloworld/ ❶
    com/javamodularity/helloworld/
      HelloWorld.java
    module-info.java ❷
```

❶ module directory

❷ module descriptor

Compared to the traditional layout of Java source files there are two major differences. First, there is an extra level of indirection: below `src` we introduce another directory `helloworld`. By convention, the directory is named after the name of the module we're creating. Second, inside this module directory we find both the source file (nested in its package structure as usual) and a *module descriptor*. A module descriptor lives in module-info.java and is the key ingredient to Java modules. Its presence signals the Java compiler we are working with a module rather than plain Java sources. Compiler behavior is quite different between working with modules and plain Java source files, as we will see in the remainder of this chapter. The module descriptor must be present in the root of the module directory. It is compiled along with the other source files into a binary class-file called module-info.class.

So what's inside the module descriptor? Our *Modular Hello World* example is quite minimalistic:

```
module helloworld {

}
```

We declare a module by using the new `module` keyword, followed by the module name. The name *must* match the name of the directory containing the module descriptor. Otherwise, the compiler refuses to compile and reports the mismatch.

This is only true when running the compiler in multi-module mode, which is the most common scenario. For the single-module scenario discussed in "Compilation" on page 36 the directory name does not matter, as long as the compilation output directory is named correctly. In any case, it's still a good idea to use the name of the module as directory name.

Because of the empty module declaration body, nothing from the helloworld module is exposed to other modules. Even though there's no dependency information (yet) in

this declaration, remember that this module implicitly depends upon the `java.base` platform module.

Module names live in a global namespace separate from other namespaces in Java. So, theoretically, you could give a module the same name as a class, interface, or package. In practice, this might lead to confusion. A good middle ground can be to take the root of the contained packages names as module name. For example, a module containing packages `com.javamodularity.app.ui` and `com.javamodularity.app.domain` would be called `com.javamodularity.app`.

Module names must be unique. Therefore, it is best to use the reverse-domain naming convention that has been popularized for JAR files and packages as well. If we want our helloworld module to be globally useable, a better name is for example `com.javamodularity.helloworld`. In this book, however, we prefer shorter module names to increase the readability of examples.

You may be wondering if adding a new keyword to the language breaks existing code that uses `module` as identifier. Fortunately, that's not the case. You can still use identifiers called `module` in your other source files, because the `module` keyword is a *contextual keyword*. It is only treated as keyword inside module-info.java. The same holds for the `requires` keyword and other new keywords we've seen so far in module descriptors.

The module-info name

Normally, the name of a Java source file corresponds to the (public) type it contains. For example, our file containing the HelloWorld class must be in a file named HelloWorld.java. The module-info name breaks this correspondence. Moreover, module-info is not even a legal Java identifier since it contains a dash. This is done on purpose, to prevent non-module-aware tools from blindly processing module-info.java or module-info.class as if it were a normal Java source file.

Reserving a name for special source files is not unprecedented in the Java language. Before module-info.java, there was package-info.java. Although it is relatively unknown, it's been around since Java 1.5. In package-info.java, you can add documentation and annotations to a package declaration. Like module-info.java, it is compiled to a class file by the Java compiler.

We now have a module descriptor containing nothing but a module declaration, and a single source file. Enough to compile our first module!

Compilation

Having a module in source format is one thing, but we can't run it without compiling first. Prior to Java 9, the Java compiler is invoked with a destination directory and a set of sources to compile:

```
javac -d out src/com/foo/Class1.java src/com/foo/Class2.java
```

In practice this is often done under the hood by build tools like Maven or Gradle, but the principle remains the same. Classes are emitted in the target directory (`out` in this case) with nested folders representing the input (package) structure. Following the same pattern, we can compile our Modular Hello World example:

```
javac -d out/helloworld src/helloworld/com/javamodularity/helloworld/
HelloWorld.java src/helloworld/module-info.java
```

There are two notable differences:

- We output into a `helloworld` directory, reflecting the module name.
- We add module-info.java as an additional source file to compile.

The presence of `module-info.java` in the set of files to be compiled triggers the module-aware mode of javac. Running the compilation results in the following output, also known as the *exploded module* format:

```
out/
  helloworld/
    com/javamodularity/helloworld/
      HelloWorld.class
    module-info.class
```

It's best to name the directory containing the exploded module after the module, but not required. Ultimately, the module system takes the name of the module from the descriptor, not from the directory name. Later, in the "Running Modules" on page 38 section we'll take this exploded module and run it.

Compiling multiple modules

What we've seen so far is the so-called *single module mode* of the Java compiler. Typically, the project you want to compile consists of multiple modules. These modules may or may not refer to each other. Or, the project is a single module but uses other (already compiled) modules. For these cases, additional compiler flags have been introduced: `--module-source-path` and `--module-path`. These are the module-aware counter parts of the `-sourcepath` and `-classpath` flags that have been part of javac for a long time. Their semantics are explained when we start looking at multi-module examples in "A Tale of Two Modules" on page 43. Keep in mind the name of

the module source directories *must* match the name declared in module-info.java in this multi-module mode.

Build tools

It is not common practice to use the Java compiler directly from the command-line, manipulating its flags and listing all source files manually. More often, build tools like Maven or Gradle are used to abstract away these details. However, these tools need to adapt to the new modular reality as well. In ??? we show how some of the most popular build tools can be used to build Java 9 modules.

Packaging

So far, we've created a single module and compiled it into the exploded module format. In the next section we show how you can run such an exploded module as-is. This works in a development situation, but in production scenarios you want to distribute your module in a more convenient format. To this end, modules can be packaged and used in JAR files. This results in *modular JAR* files.

The JAR tool has been updated to work with modules in Java 9. To package up the Modular Hello World example, execute the following command:

```
jar -cfe mods/helloworld.jar com.javamodularity.helloworld.HelloWorld -C
out/helloworld .
```

With this command, we're creating a new archive (`-cf`) called `helloworld.jar` in the `mods` directory (make sure the directory exists, though). Furthermore, we want the entry point (`e`) for this module to be the HelloWorld class. Meaning, whenever the module is started without specifying another main class to run, this is the default. We provide the fully qualified classname as an argument for the entry point. Finally, we instruct the jar tool to change (`-C`) to the `out/helloworld` directory and put all compiled files from this directory in the JAR file. The contents of the JAR are now similar to exploded module, with the addition of a `MANIFEST.MF` file:

```
helloworld.jar
  META-INF/
    MANIFEST.MF
  com/javamodularity/helloworld/
    HelloWorld.class
  module-info.class
```

The name of the modular JAR file is not significant, unlike the situation with the module directory name. You can use any filename you like, since the module is identified by the name declared in the bundled module-info.class.

Running Modules

Let's recap what we did so far. We started our Modular Hello World example by creating a helloworld module with a single HelloWorld.java source file and a module descriptor. Then, we compiled the module into the exploded module format. Finally, we took the exploded module and packaged it as a modular JAR file. This JAR file contains our compiled class, the module descriptor and knows about the main class to execute.

Now let's try to run the module. Both the exploded module format and modular JAR file can be run. The exploded module format can be started with the following command:

```
$ java --module-path out --module helloworld/com.javamodularity.helloworld.HelloWorld
Hello Modular World!
```

You can also use the short-form `-p` flag instead of `--module-path`. The `--module` flag can be shortened to `-m`.

The java command has gained new flags to work with modules rather than with *legacy* classpath-based applications. Notice we put the `out` directory (containing the exploded helloworld module) on the module path. The module path is the module-aware counterpart of the original classpath.

Next, we provide the module to be run with the `--module` flag. In this case, it consists of the module name followed by a slash and then the class to be run. On the other hand, if we run our modular JAR, providing only the module name is enough:

```
$ java --module-path mods --module helloworld
Hello Modular World!
```

This makes sense, because the modular JAR knows the class to execute from its metadata. We explicitly set the entry point to `com.javamodularity.helloworld.HelloWorld` when constructing the modular JAR.

Launching in either of these two ways makes helloworld the *root module* for execution. The JVM starts from this root module, and resolves any other modules necessary to run the root module from the module path. Resolving modules is a transitive process: if a newly resolved modules requires other modules, the module system automatically takes this into account. Since module dependencies are guaranteed to be non-circular, this is process will finish when all relevant modules are resolved.

In our simple helloworld example, there is not too much to resolve. You can trace the actions taken by the module system by adding `-Xdiag:resolver` to the java command:

```
$ java -Xdiag:resolver --module-path mods --module helloworld
[Resolver] Root module helloworld located
[Resolver]   (jar:file:///chapter3/helloworld/mods/helloworld.jar!/)
[Resolver] Module java.base located, required by helloworld
[Resolver]   (jrt:/java.base)
[Resolver] Resolver completed in 61 ms
[Resolver]   helloworld
[Resolver]   java.base
```

Options starting with `-X` are non-standard, hence may not be supported on Java implementations that are not based on OpenJDK.

In this case, no other modules are necessary (besides the platform module `java.base`) to run helloworld. An error would be encountered at startup if another module were necessary to run helloworld and it's not present on the module path. This form of reliable configuration is a huge improvement over the old classpath situation. Previously, a missing dependency is only noticed when the JVM tries to load a non-existent class at run-time. The only way for this occur with the module path, is when modules use reflection. Reflective use of a class from another module is not necessarily recorded in the module descriptor, in which case the resolver cannot prevent bad things from happening.

Module path

Even though the module path looks quite similar to the classpath, they behave differently. The module path is a list of paths to individual modules and directories containing modules. Each directory on the module path can contain zero or more module definitions, where a module definition can be an exploded module or a packaged module. An example module path containing all three different options looks like this: `out/:myexplodedmodule/:mypackagedmodule.jar`. All modules from within the `out` directory are on the module path, in conjuction with the module `myexplodedmodule` (a directory) and `mypackagedmodule` (a modular JAR file).

Entries on the module path are separated by the default platform separator. On Linux/OS X that's a colon (`java -mp dir1:dir2`), on Windows use a semi-colon (`java -mp dir1;dir2`).

Most importantly, all artifacts on the module path have module descriptors (possibly synthesized on the fly, as we will learn in ???). The resolver relies on this information to find the right modules on the module path. When multiple modules with the same name are in the same directory on the module path, the resolver shows an error and won't start the application. Again, this prevents scenarios with conflicting JAR files that were previously possible on the classpath.

Linking Modules

In the previous section we saw the module system only resolved two modules: `hello world` and `java.base`. Wouldn't it be great if we could take advantage of this upfront knowledge by creating a special distribution of the Java runtime containing the bare minimum to run our application? That's exactly what you can do in Java 9 with *modular runtime images*.

An optional linking phase is introduced with Java 9, in-between the compilation and run-time phase. With a new tool called `jlink` you can create a runtime image containing only the necessary modules to run an application. Using the following command we create a new runtime image with the helloworld module as root:

```
$ jlink --module-path mods/:$JAVA_HOME/jmods --add-modules helloworld --
output helloworld-image
```

The first option constructs a module path containing the mods directory (where helloworld lives) and the directory of the JDK installation containing the platform modules we want to link into the image. Unlike with `javac` and `java` you have to explicitly add platform modules to the jlink module path. Then, `--add-modules` indicates helloworld is the root module that needs to be runnable in the runtime image. It's possible to provide multiple root modules separated by comma (multiple `--add-modules` with a single module name also works). Each of these root modules will be resolved and these root modules along with their resolved dependencies become part of the image. Last, `--output` indicates a directory name for the runtime image.

The result of running this command is a new directory containing a Java Runtime Environment completely tailored to running helloworld:

```
helloworld-image/
  bin/
    helloworld ❶
    java ❷
    keytool
  conf/
    ...
  lib/
    ...
```

❶ An executable directly launching the helloworld module.

❷ The java runtime, capable of resolving only helloworld and its transitive dependencies.

Since the resolver knows only `java.base` is necessary in addition to helloworld, nothing more is added to the runtime image. Therefore, the runtime image is many times smaller than a full JDK. An additional `--strip-debug` flag can further reduce the runtime image size. A custom runtime image can be used on resource-constrained devices, or serve as the basis for a container image for running an application in the cloud.

If you inspect the `bin` directory of the generated runtime image, you'll find a `helloworld` executable. This executable is convenience wrapper that runs the JVM directly with the correct module path and helloworld as root module. Such an executable is created for every root module listed with `--add-modules`, given it has an entry point (main class) like we specified for `helloworld.jar`. You can also build a runtime image from exploded modules. In that case, no binary will be created since the main class is not identifiable by the jlink tool. Instead, you can invoke `bin/java -m helloworld/com.javamodularity.helloworld.HelloWorld` to start the application. Note that there's no need to set the module path when running the `java` command from the runtime image. All necessary modules are already there by virtue of the linking process.

We can show the runtime image indeed contains the bare minimum of modules by running `java --listmodules`:

```
$ helloworld-image/bin/java --list-modules
helloworld
java.base@9
```

The bin directory can contain other executables besides the ones discussed so far. For the helloworld example, the JDK `keytool` binary is also added. Since other well-known JDK tools like `jar`, `rmic` and `javaws` all use modules that are not in this runtime image, jlink is smart enough to omit them.

Of course, the same caveats with respect to reflection apply to linking as with running modules. If code uses reflection and does not have those dependencies in the module descriptor, the resolver cannot take this into account. In that case, you must add the modules manually to the `--add-modules` argument, so they end up in the runtime image anyway.

No Module Is an Island

So far, we've purposely kept things small and simple in order to understand the mechanics of module creation and usage. However, the real magic happens when you

compose multiple modules. Only then the advantages of the module system become apparent.

It would be rather boring to extend our Modular Hello World example. Therefore, we continue with a more interesting example application called EasyText. Starting from a single monolithic module, we gradually create a multi-module application. EasyText may not be as big as your typical enterprise application (fortunately), but it touches enough real-world concerns to serve as a learning vehicle.

Introducing the EasyText Example

EasyText is an application for analyzing text complexity. It turns out there are quite some interesting algorithms you can apply to text to determine its complexity. Read "Text complexity in a nutshell" on page 42 for more details.

Of course, our focus is not on the text analysis algorithms, rather on the composability of the modules making up EasyText. The goal is to use Java modules to create a flexible and maintainable application. Here are the requirements we want to fulfill through a modular implementation of EasyText:

- It has the ability to add new analysis algorithms without modifying or recompiling existing modules.
- Different front-ends (for example GUI and command-line) are able to use the same analysis logic.
- It must support different configurations, without recompilation and without deploying all code for each configuration.
- It uses existing (possibly non-modularized) libraries where necessary.

Granted, all of these requirements *can* be met without modules. It's not an easy job though. Using the Java module system helps us to meet these requirements.

Text complexity in a nutshell

Even though the focus of the EasyText example is on the structure of the solution, it never hurts to learn something new along the way. Text analysis is a field with a long history. The EasyText example applies *readability formulas* to texts. One of the most popular readability formulas is the Flesch-Kincaid test:

$$complexity_{flesch\_kincaid} = 206.835 - 1.015\frac{totalwords}{totalsentences} - 84.6\frac{totalsyllables}{totalwords}$$

Given some relatively easily derivable metrics from a text, a score is calculated. If a text scores between 90 and 100, it is easily understood by an average 11-year-old stu-

dent. Texts scoring in the range between 0 and 30 on the other hand are best suited to graduate level students.

There are numerous other readability formulas, such as Coleman-Liau, Fry readability, not to mention the many localized formulas. Each formula has its own scope and there is no single best one. Of course this is one of the reasons to make EasyText as flexible as possible.

Throughout this chapter and the subsequent chapters each of these requirements is addressed. From a functional perspective, analyzing a text comprises several steps:

1. Input the text (either through file, GUI or otherwise),
2. Split up the text into sentences and words (since many readability formulas work with sentence or word-level metrics),
3. Run one or more analyses on the text,
4. Show the result to the user.

Initially our implementation consists of a single module, `easytext`. With this starting point, there is no separation of concerns. There's just a single package inside a module, which by now we are familiar with:

```
src/
  easytext/
    javamodularity/easytext/
      Main.java
    module-info.java
```

The module descriptor is an empty one. The Main class reads a file, applies a single readability formula (Flesch-Kincaid) and prints the results to the console. After compiling and packaging the module it works like this:

```
$ java --module-path out/mods -m easytext input.txt
Reading input.txt
Flesh-Kincaid: 83.42468299865723
```

Obviously, the single module setup ticks none of the boxes as far as our requirements are concerned. It's time to add more modules into the mix.

A Tale of Two Modules

As a first step, we separate the text analysis algorithm and the main program into two modules. This opens up the possibility to reuse the analysis module with different front-end modules later on. The main module uses the analysis module, as shown in Figure 3-1.

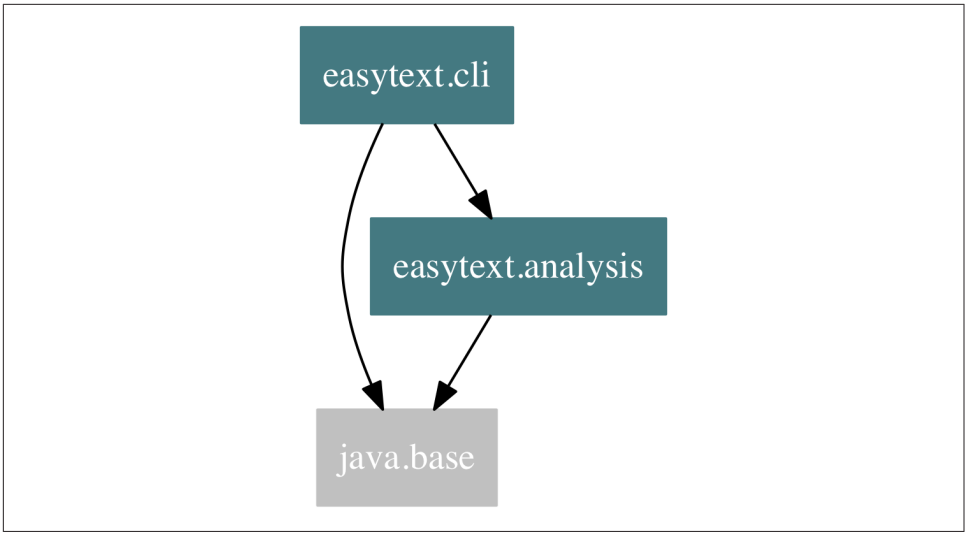

Figure 3-1. EasyText in two modules

The `easytext.cli` module contains the command-line handling and file parsing code. The `easytext.analysis` module contains the implementation of the Flesch-Kincaid algorithm. During the split of the single `easytext` module, we create two new modules with two different packages:

```
src/
  easytext.cli/
    javamodularity/easytext/cli/
      Main.java
    module-info.java
  easytext.analysis/
    javamodularity/easytext/analysis/
      FleschKincaid.java
    module-info.java
```

The difference is the Main class now delegates the algorithmic analysis to the Flesch-Kincaid class. Since we have two modules, we need to compile them using the multi-module mode of javac:

```
javac -d out --module-source-path src <list of all Java source files>
```

From this point onward, we assume modules of our examples are always compiled together, with the `--module-source-path` flag and by listing all source-files[11]. This flag tells `javac` where to look for other modules in source format during compilation. It is mandatory to provide a destination directory with `-d` when compiling in multi-

1 On a Linux/OS X system you can easily provide `$(find . -name '*.java')` as last argument to achieve this.

module mode. After compilation, the destination directory contains the compiled modules in exploded module format. This output directory can then be used as element on the module path when running modules.

In this example, `javac` looks up `FleschKincaid.java` on the module source path when compiling `Main.java`. But how does the compiler know to look in the `easytext.analysis` module for this class? In the old classpath situation, it might have been in any JAR that is put on the compilation classpath. Remember, the classpath is a flat list of types. Not so for the module path, it only deals with modules. Of course the missing piece of the puzzle is in the content of module-info descriptors. They provide the necessary information for locating the right module exporting a given package. No more aimless scanning of all available classes wherever they live.

In order for the example to work, we need to express the dependencies we've already seen in Figure 3-1. The analysis module needs to expose the FleschKincaid class:

```
module easytext.analysis {
   exports javamodularity.easytext.analysis;
}
```

With the `exports` keyword, packages in the module are exposed for use by other modules. By declaring that package `javamodularity.easytext.analysis` is exported, all its public types can now be used by other modules. A module can export multiple packages. In this case, only the FleschKincaid class is exposed to other modules. Conversely, every package inside a module that is *not* exported, is private to the module.

We've seen how the analysis module exposes the FleschKincaid class. The module descriptor for `easytext.cli`, on the other hand, needs to express its dependency on the analysis module:

```
module easytext.cli {
   requires easytext.analysis;
}
```

We require the module `easytext.analysis` because the Main class imports the FleschKincaid class, originating from that module. With both these module descriptors in place, the code compiles and can be run.

What happens if we omit the requires statement from the module descriptor? In that case, the compiler produces the following error:

```
./src/easytext.cli/javamodularity/easytext/cli/Main.java:11:
  error: FleschKincaid is not visible because package
  javamodularity.easytext.analysis is not visible
import javamodularity.easytext.analysis.FleschKincaid;
                                        ^
```

Even though the FleschKincaid.java source file is still available to the compiler, it throws this error. Exactly the same error is produced when we omit the exports statement from the analysis module's descriptor. Here we see the major advantage of making dependencies explicit in every step of the software development process. A module can only use what it requires, and the compiler enforces this. At runtime, the same information is used by the resolver to ensure all modules are present before starting the application. No more accidental dependencies on libraries, only to find out at runtime this library isn't available on the classpath.

Another check the module system enforces is for cyclic dependencies. In the previous chapter we've learned that readability relations between modules must be acyclic at compile-time. Within modules, you can still create cyclic relations between classes as has always been the case. It's debatable whether you really want to do so from a software engineering perspective, but you can. However, at the module level, there is no choice. Dependencies between modules must form an acyclic, directed graph. By extension, there can never be cyclic dependencies between classes in different modules. If you do introduce a cyclic dependency, the compiler won't accept it. Adding `requires easytext.cli` to the analysis module descriptor introduces a cycle, as shown in Figure 3-2.

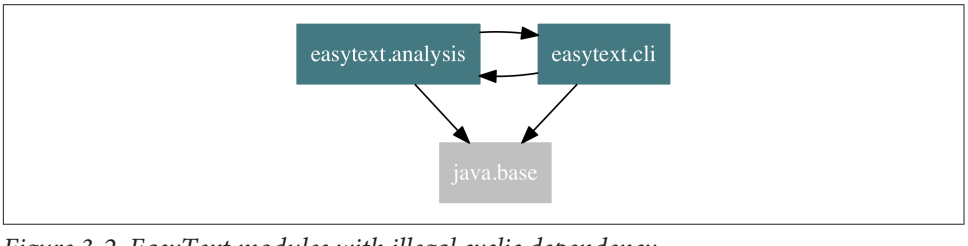

Figure 3-2. EasyText modules with illegal cyclic dependency.

If you try to compile this, you run into the following error:

```
./src/easytext.cli/module-info.java:2:
   error: cyclic dependence involving easytext.analysis
   requires easytext.analysis;
```

Note that cycles can be indirect as well, as illustrated in Figure 3-3. These cases are less obvious in practice, but are treated the same as direct cycles: they result in an error from the Java module system.

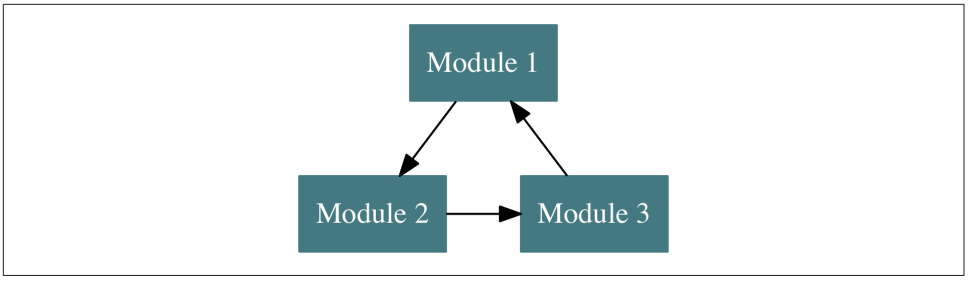

Figure 3-3. Cycles can be indirect as well.

Many real-world applications do have cyclic dependencies between their components. In chapter ??? we discuss how to prevent and break cycles in your application's module graph.

Working with Platform Modules

Platform modules come with the Java runtime and provide functionality like XML parsers, GUI toolkits and other libraries that you expect to see in a Java runtime. In Figure 2-1 we've already seen a subset of the platform modules. From a developers perspective they behave the same as application modules. Platform modules encapsulate certain code, possibly export packages and can have dependencies on other (platform) modules. Having a modular JDK means you need to be aware of what platform modules you're using in application modules.

In this section we extend the EasyText application with a new module. It's going to use platform modules, unlike the modules we've created thus far. Technically we did use a platform module already: the `java.base` module. However, this is an implicit dependency. The new module we are going to create has explicit dependencies on other platform modules.

Finding the Right Platform Module

If you need to be aware of the platform modules you use, how do you find out which platform modules exist? You can only depend on a (platform) module if you know its name. When you run `java --list-modules`, the runtime outputs all available platform modules.

```
$ java --list-modules
java.base@9
java.xml@9
javafx.base@9
jdk.compiler@9
jdk.management@9
```

This abbreviated output shows there are several types of platform modules. Platform modules prefixed with `java.` are part of the Java SE specification. They export APIs as standardized through the Java Community Process for Java SE. The JavaFX APIs are distributed in modules starting with `javafx.`. Modules prefixed with `jdk.` contain JDK specific code which may be different across JDK implementations.

Even though the `--list-modules` functionality is a good starting point for discovering platform modules, you need more. Whenever you import from a package that's not exported by `java.base` you need to know which platform module provides this package. That package must be added to module-info.java with a `requires` clause. So let's return to our example application to find out what working with platform modules entails.

Creating a GUI Module

EasyText so far has two application modules working together. The command-line main application has been separated from the analysis logic. In the requirements we stated that we want to support multiple front-ends on top of the same analysis logic. So let's try to create a GUI front-end in addition to the command-line version. Obviously, it should reuse the analysis module that is already in place.

We're going to use JavaFX to create a modest GUI for EasyText. As of Java 8, the JavaFX GUI framework has been part of the Java platform and is intended to replace the older Swing framework. The GUI looks like this:

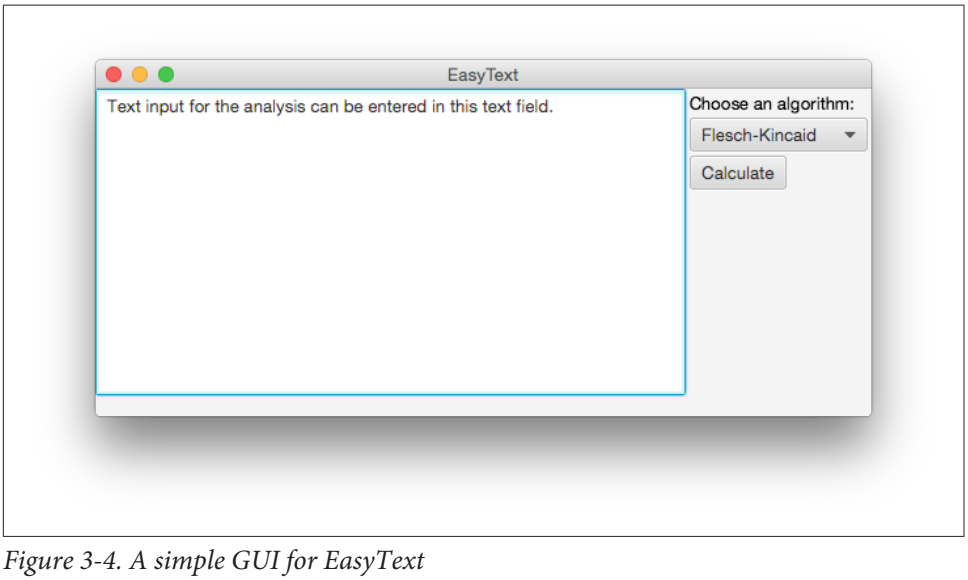

Figure 3-4. A simple GUI for EasyText

When you click Analyze, the analysis logic is run on the text from the textfield and the resulting value is shown in the GUI. Currently, we only have a single analysis algorithm that can be selected in the dropdown, but that will change later on given our extensibility requirements. For now we'll keep it simple and assume the Flesch-Kincaid analysis is the only available algorithm. The code for the GUI Main class is quite straightforward:

```
package javamodularity.easytext.gui;

import java.util.ArrayList;
import java.util.List;

import javafx.application.Application;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.text.Text;
import javafx.stage.Stage;

import javamodularity.easytext.analysis.FleschKincaid;

public class Main extends Application {

    private static ComboBox<String> algorithm;
    private static TextArea input;
    private static Text output;

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("EasyText");
        Button btn = new Button();
        btn.setText("Calculate");
        btn.setOnAction(event ->
          output.setText(analyze(input.getText(), (String) algorithm.getValue()))
        );

        VBox vbox = new VBox();
        vbox.setPadding(new Insets(3));
        vbox.setSpacing(3);
        Text title = new Text("Choose an algorithm:");
        algorithm = new ComboBox<>();
        algorithm.getItems().add("Flesch-Kincaid");

        vbox.getChildren().add(title);
        vbox.getChildren().add(algorithm);
```

```
            vbox.getChildren().add(btn);

            input = new TextArea();
            output = new Text();
            BorderPane pane = new BorderPane();
            pane.setRight(vbox);
            pane.setCenter(input);
            pane.setBottom(output);
            primaryStage.setScene(new Scene(pane, 300, 250));
            primaryStage.show();
        }

        private String analyze(String input, String algorithm) {
            List<List<String>> sentences = toSentences(input);

            return "Flesch-Kincaid: " + new FleschKincaid().analyze(sentences);
        }

        // implementation of toSentences() omitted for brevity
    }
```

There are imports from eight different JavaFX packages in the the Main class. How do we know which platform modules to require in module-info.java? One way to find out in which module a package lives is through JavaDoc. For Java 9, JavaDoc has been updated to include the module name that each type is part of.

Another approach is to inspect the JavaFX modules available using `java --list-modules`. After running this command we see eight modules containing `javafx` in the name:

```
javafx.base@9
javafx.controls@9
javafx.deploy@9
javafx.fxml@9
javafx.graphics@9
javafx.media@9
javafx.swing@9
javafx.web@9
```

Since there is not always a one-to-one correspondence between the module name and the packages it contains, choosing the right module is somewhat of a guessing game from this list. You can inspect the module declarations of platform modules to verify assumptions. If, for example, we think `javafx.controls` might contain the `javafx.scene.control` package, we can verify that with:

```
$ java --list-modules javafx.controls
javafx.controls@9
  requires public javafx.base
  requires public javafx.graphics
  requires mandated java.base
  exports com.sun.javafx.scene.control to javafx.web
```

```
exports com.sun.javafx.scene.control.behavior to javafx.web
exports com.sun.javafx.scene.control.inputmap to javafx.web
exports com.sun.javafx.scene.control.skin to javafx.web, javafx.graphics, javafx.deploy
exports javafx.scene.chart
exports javafx.scene.control ❶
exports javafx.scene.control.cell
exports javafx.scene.control.skin
conceals com.sun.javafx.charts
conceals com.sun.javafx.scene.control.skin.resources
```

❶ Module `javafx.controls` exports the `javafx.scene.control` package

Indeed, the package we want is contained in this package. This process of manually finding the right platform module is a bit tedious. It's expected that IDE's will support the developer with this task once Java 9 support is in place. For the EasyText GUI, it turns out we need to require two JavaFX platform modules:

```
module easytext.gui {
   requires javafx.graphics;
   requires javafx.controls;
   requires easytext.analysis;
}
```

Given this module descriptor the GUI module compiles correctly. However, when trying to run it, the following curious error comes up:

```
Exception in thread "main" java.lang.RuntimeException:
   Unable to construct Application instance:
class javamodularity.easytext.gui.Main
  at ...LauncherImpl.launchApplication1(javafx.graphics@9/LauncherImpl.java:926)
  at ...lambda$launchApplication$140(javafx.graphics@9/LauncherImpl.java:220)
  at java.lang.Thread.run(java.base@9/Thread.java:804)
Caused by: java.lang.IllegalAccessException:
  class com.sun.javafx.application.LauncherImpl (in module javafx.graphics)
  cannot access class javamodularity.easytext.gui.Main (in module
  javamodularity.easytext.gui) because module javamodularity.easytext.gui
  does not export javamodularity.easytext.gui to module javafx.graphics
```

What is going on here? By extending the `javafx.application.Application` class (which lives in the `javafx.graphics` module) and calling `launch()` from the main method, we delegate the bootstrapping of the GUI to the JavaFX framework. JavaFX then uses reflection to instantiate Main, subsequently invoking the `start()` method. For this to work, the `javafx.graphics` bundle must have a readability relation to the package containing our Main class.

Fortunately, the module system is smart enough to dynamically establish the readability relation when reflection is used to our GUI module. The problem is, the package containing Main is never exposed from the GUI module. Main is not accessible for the `javafx.graphics` module. This is exactly what the error message above tells us.

One solution would be adding an `exports` clause for the `javamodularity.easytext.gui` package to the module descriptor. Only, that would expose the Main class to any module requiring the GUI module. Is that really what we want? Is the Main class really part of a public API we want to expose? Not really. The only reason we need it to be accessible is because JavaFX needs to instantiate it. This is a perfect use-case for qualified exports:

```
module easytext.gui {

   exports javamodularity.easytext.gui to javafx.graphics;

   requires javafx.graphics;
   requires javafx.controls;
   requires easytext.analysis;
}
```

During compilation, the target modules of a qualified export must be on the module path or be compiled at the same time. Obviously this is not an issue for platform modules, but it is something to be aware when using qualified exports to non-platform modules.

Through the qualified export, only `javafx.graphics` is able to access our Main class. Now we can run the application and JavaFX is able to instantiate Main.

An interesting situation arises at run-time. The `javax.graphics` module dynamically establishes a readability relation with `easytext.gui` at run-time (depicted by the bold edge in Figure 3-5).

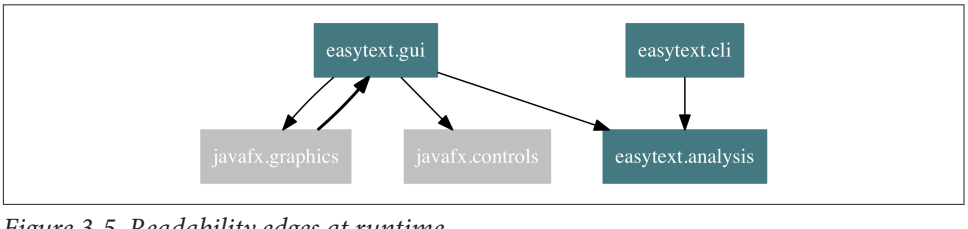

Figure 3-5. Readability edges at runtime.

But that means there is cycle in the readability graph! How can this be? Cycles were supposed to be impossible. They are, *at compile time*. In this case, we compile the `easytext.gui` with a dependency on (and thus readability relation to) `javafx.graphics`. At run-time, `javax.graphics` automatically establishes a readability relation to `easytext.gui` when it reflectively instantiates `Main`. Readability relations can be cyclic *at runtime*. Because the export is qualified, only `javafx.graphics` can access our Main class. Any other module establishing a readability relation with `easytext.gui` won't be able to access the `javamodularity.easytext.gui` package.

The Limits of Encapsulation

Looking back, we have come a long way in this chapter. We learned how to create modules, run them and use them in conjuction with platform modules. Our example application, EasyText, has grown from a *mini-monolith* to a multi-module Java application. Meanwhile achieving two of the stated requirements: it supports multiple front-ends while reusing the same analysis module, and we can create different configurations of our modules targeting the command-line or a GUI.

Looking at the other requirements, however, there's still a lot to be desired. As things stand, both front-end modules instantiate a specific implementation class (`FleschKincaid`) from the analysis module to do their work. Even though the code lives in separate modules, there is tight coupling going on here. What if we want to extend the application with different analyses? Should every front-end module be changed to know about new implementation classes? That sounds like a poor encapsulation. Should the front-end modules be updated with dependencies on newly introduced analysis modules? That sounds distinctly non-modular. It also runs counter to our requirement of adding new analysis algorithms without modifying or recompiling existing modules. Figure 3-6 already shows how messy this gets with two front-ends and two analyses (Coleman-Liau is another well-known complexity metric.)

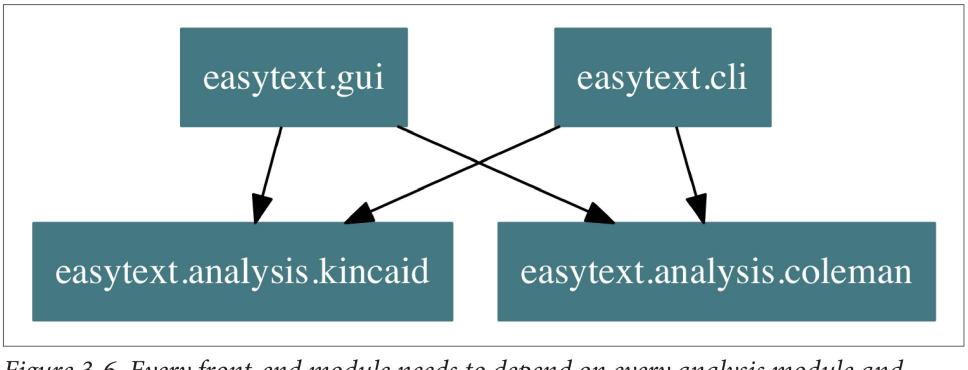

Figure 3-6. Every front-end module needs to depend on every analysis module and instantiate its exported implementation class.

In summary, we have two issues to address:

1. The front-ends need to be de-coupled from concrete analysis implementation types and modules. Analysis modules should not export these types either to avoid tight coupling.
2. Front-ends should be able to *discover* new analysis implementations in new modules without any changes.

By solving these two problems, we satisfy the requirement that new analyses can be added by just adding them on the module path, without touching the front-ends.

Interfaces and Instantiation

Ideally, we'd abstract away the different analyses behind an interface. After all, we're just passing in sentences and getting back a score for each algorithm:

```
public interface Analyzer {

   double analyze(List<List<String>> text);

}
```

As long as we can find out the name of an algorithm (for display purposes) and get it to calculate the complexity, we're good. This type of abstraction is what interfaces were made for. The Analyzer interface is stable and can live in its own module, say `easytext.analysis.api`. That's what the front-end modules should know and care about. The analysis modules require this API module as well and implement the Analyzer interface. So far, so good.

However, there's a problem. Even though the front-end modules only care about calling the `analyze()` method through the Analyzer interface, they still need a concrete instance to call this method on:

```
Analyzer analyzer = ???
```

How can we get a hold of an instance that implements Analyzer without relying on a specific constructor? You can say:

```
Analyzer analyzer = new FleschKincaid();
```

but that brings us right back to square one. Instead, we need a way to obtain instances without referring to concrete implementation classes. Like with all problems in computer science, we can try to solve this by adding a new layer of indirection. We will look at solutions for this problem in the next chapter, where we learn about the *Factory Pattern* and *Services*.