Chapter A

APPENDIX A.

JDK 11 was released on the 25th of September. Aside all novelties already covered in this book, there was also the news developers were suspecting since the introduction of modules. Java is still free, but the Oracle JDK isn't. Developers can use the Oracle JDK to learn, but applications that will be deployed in production will require a license. The workaround is to use the an OpenJDK build(from here: `https://jdk.java.net/11/`), but this does not come with Oracle support. Still JDK 11 is a major release that will benefit from long term support and if you are curious about the Oracle terms of use, you can read them here: `https://www.oracle.com/technetwork/java/javase/terms/license/javase-license.html`

The Java world has changed, but Oracle restrictions will only cause the market to provide more options. Companies like Zulu, IBM will most probably develop their own JDKs and will provide support for JDKs Oracle won't. Probably more companies will emerge to provide cheaper support and open source JDKs. Now that access to an important resource has been restricted, humanity will do what it does best - become creative, to either get access to it, or develop similar resources. Either way, diversity will be blooming in the following years for the Java Open Source Community, and I can barely wait to see what is coming.

The purpose of this Appendix is to gracefully end the book and to cover more advanced details regarding Java modules, that might not be suitable for a Java beginner developer right at the start of the book. It contains an extended version of **Chapter 3**, covering configuration of Java modules in a complex project, good, bad and recommended practices working with modules and the explanation for the multi-module Gradle structure of the project. The code snippets mentioned in this appendix are already part of the project associated with the book. Enjoy!

A.1 Chapter 3: Introducing modules (advanced version)

Starting with Java 9 a new concept was introduced: `modules`, that will be used to group together and encapsulate packages. Implementation of this new concept took more than 10 years. The discussion about modules started in 2005 and it was proposed to be implemented for Java 7. Under the name **Project Jigsaw** an exploratory phase started in 2008. Java developers hoped a modular JDK will be available with Java 8, but only in Java 9 it was made possible, after three years of work (and almost 7 of analysis). Apparently this is why the official release date for Java 9 was postponed to September 2017. [1]

Modules represent a new way to aggregate packages. A `module` is a way to group them together and configure more granulated access to package contents.

A `module` is a uniquely named, reusable group of packages and resources (XML files) described by a file named `module-info.java`. This file contains the following information:

- the module's name
- the module's dependencies (that is, other modules this module depends on)

[1] The full history of the Jigsaw project can be found here: `http://openjdk.java.net/projects/jigsaw/`

- the packages it explicitly makes available to other modules (all other packages in the module are implicitly unavailable to other modules)
- the services it offers
- the services it consumes
- to what other modules it allows reflection
- native code
- resources
- configuration data

In theory, module naming resembles package naming and follows the reversed-domain-name convention. In practice, just make sure the module name does not contain any numbers and that it reveals clearly what its purpose is. The `module-info.java` file is compiled into a `module descriptor`, which is a file named `module-info.class` that is packed together with classes into a plain old JAR file. The location of the file is in the root sources directory, outside of any package. For the example introduced earlier, a `module-info.java` was added and the new project structure is depicted in Figure A.1.

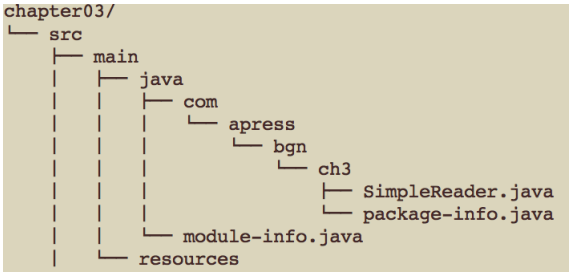

Figure A.1: Structure of a Java 9 project

As any file with with `*.java` extension, the `module-info.java` get compiled into a `*.class` file. As the module declaration is not a part of Java object types declaration, `module` is not a Java keyword, so it can still be used when writing code for Java object types. For `package` the situation is different, as every Java object type declaration must start with a package declaration. Just take a look at the `SimpleReader` class declared below.

```
package com.apress.bgn.ch3;

public class SimpleReader {

    private String source;
    ...
}
```

So what does this actually mean? Where actually is the module and what is it? Well, in simple projects, that are made of one root directory with sources, modules do not have to physically delimit or organize sources somehow [2]. They are just defined by the contents of the `module-info.java` file. So, starting with Java 9 Figure 3.4(from the short version of **Chapter 3**) evolves into Figure A.2

[2]Unless you rename directories containing sources for a module to the module name that is. Having actual directories for modules is unavoidable, when the sources in the root directory of a project must be split into different modules.

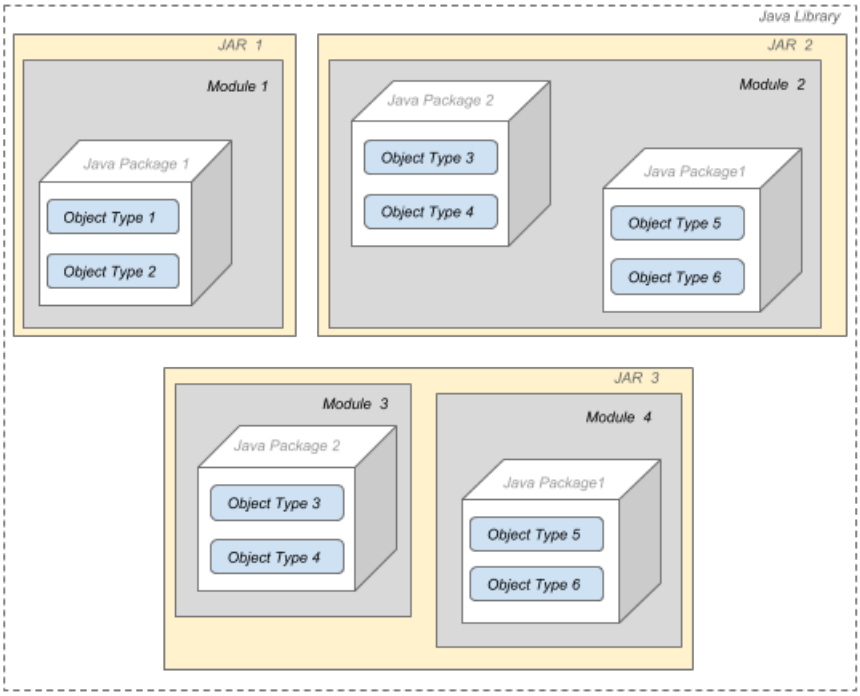

Figure A.2: Java building blocks, starting with Java 9

In the previous Figure, for JAR1 and JAR2, there is no need to create a directory for the module. For JAR3, we have the case of two modules being archived in the same JAR, and in this case we need to explicitly separate their sources as well, and the reason for this is obviously the need to have two `module-info.java`. An example of such a project will be covered later.

The introduction of modules allows for the JDK to be divided into modules as well. This means that the Java platform is no longer a monolith, consisting of a massive number of packages, making it challenging to develop, maintain and evolve. The platform is now split into 95 modules that can be viewed by executing `java --list-modules` (the number might vary in Java later versions).

```
 $ java --list-modules
java.base@11
java.compiler@11
java.datatransfer@11
java.desktop@11
...
```

Each module name is followed by a version string `@11`, in the previous listing, which means that the module belongs to Java 11.
So, if a java application does not require all modules, a runtime can be created only with the modules that it needs, which reduces the runtime's size.The tool to build a smaller runtime customized to an application needs is called `jlink` and is part of the JDK executables. This allows for bigger levels of scalability and increased performance. [3]

Another benefit of introducing modules is stronger encapsulation which in turns leads to better integrity. Before Java 9, a lot of platform classes could be used and code could be tight to them, which lead to dependencies on legacy

[3]How to use **jlink** is not an object of this book. The focus of the book is learning the Java programming language, thus the technical details of the Java platform will be kept to a minimum, just enough to start writing and executing code confidently.

code and difficulty to migrate to another Java version. When writing Java code, you will notice that most Java object types that you will use come from packages such as `java.*...` or `javax.*....` Those packages represent the core of the Java Platform and contain classes intended for public use. But the JDK also contains other packages that contain classes that were not intended for external use, that are used to implement JDK internal behaviour, like the `sun.*` packages. Until Java 9, they could be imported and the classes within them used, just like any other package. Starting with JDK 9, the internal APIs are encapsulated and are made inaccessible by default, to prevent a developer from tying an implementation to a specific JDK version.

Let's skip ahead and define a module, and let's go through all the configurations we might need for one. But before that, let's add one more package to the `chapter03` project so we have enough material to play with, so the project structure now is depicted in Figure A.3.

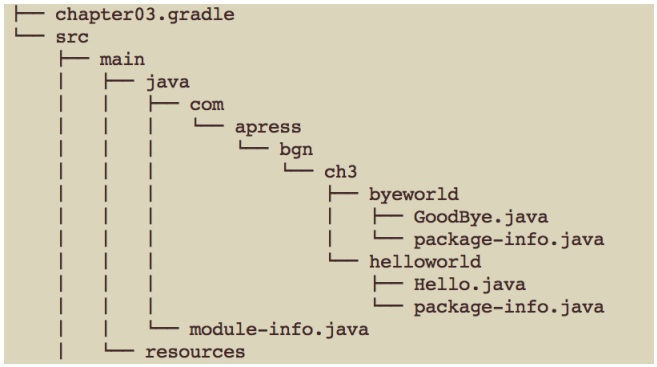

Figure A.3: Java 9 project with two packages and one module

The contents of the `module-info.java` can be as simple as the name of the module and two brackets.

```
module chapter.three {
}
```

How to configure modules

Within those brackets, different `module directives` may be declared, using one of the following keywords:

- `requires`
- `exports`
- `module`
- `open`
- `opens... to`
- `provides ... with`
- `transitive`

We'll go over all of them in this section and give examples, so that when you will start writing java code, you will know how to declare your modules correctly.

Modules can have dependencies. For our example classes inside the module `chapter.three` need access to packages and classes in module `chapter.zero`. Declaring a module dependency is done my using the `requires` keyword.

```
module chapter.three {
    requires chapter.zero;
}
```

The dependency above is an **explicit** one. But there are also implicit dependencies. For example, any module declared by a developer **implicitly** requires the `java.base` module. This module defines the foundational APIs of the Java SE Platform, and no Java application could be written without it.

Declaring a module as required, means that that module is required at compile time and runtime. If a module is required only at runtime, the `requires static` keywords are used to declare the dependency. Just keep that in mind for now, it will make sense when we talk about web applications.

But is it enough to declare our module as dependent of another? Does this mean that the dependent module can access all `public` types(and their nested `public` and `protected` types) ? If you are thinking not, you are right. This is of course, because the module it depends on, must be configured to actually expose its *insides*. How can that be done? In our case, we need to make sure module `chapter.zero` gives access to the required packages. This is done by customizing the `module-info.java` for this module by adding the `exports` directive, followed by the necessary package names.

```
module chapter.zero {
    exports com.apress.bgn.ch0;
}
```

By doing this we have just given access to the `com.apress.bgn.ch0` package, to any module that will require this module as a dependency. What if we do not want that?

! If you were curious and read the recommended **Jar Hell** article, you noticed that one of the concerns of working with Java sources packed in Jars, was security. Because, even without access to Java sources, by adding a Jar as a dependency to an application, objects could be accessed, extended and instantiated. So, aside from providing a reliable configuration, better scaling, integrity for the platform and improved performance, the goal for introduction of modules was better security as well.

What if we want to **limit the access to module contents** only to the `chapter.three` module? This can be done by adding the `to` keyword followed by the module name to the `exports` directive.

```
module chapter.zero {
    exports com.apress.bgn.ch0 to chapter.three;
}
```

More than one module can be specified to have access, by listing the desired modules, separated by comma.

```
module chapter.zero {
    exports com.apress.bgn.ch0 to chapter.three, chapter.two;
}
```

All good and well, but we can go even one step further. What if module `chapter.three` requires access to a class defined in a module that is a dependency of `chapter.zero`? In technical language this is called a **transitive dependency**, because it is obviously more practical to use a dependency that is already there, instead of declaring it again. Modules support this as well and the term to declare such a dependency is(as you probably suspected): `transitive`.

For this scenario, we'll make our module `chapter.zero` depend on external module of the LOG4J(Apache

Log4j 2), that is a simple library for logging of application behavior.[4] But we also want any module depending on `chapter.zero` to be able to use classes in the `org.apache.logging.log4j` module. In this case, the contents of the `module-info.java` for module `chapter.zero` becomes:

```
module chapter.zero {
    requires transitive org.apache.logging.log4j;

    exports com.apress.bgn.ch0 to chapter.three;
}
```

By using `requires transitive` we have given read access to module `org.apache.logging.log4j` to our `chapter.three`. This means that object types in `chapter.three` can de declared by making use of object types defined in packages exported by module `org.apache.logging.log4j`.

And this is where simple, and sort-of understandable things end. There are a few more module directives we have to cover that will be a little more difficult to grasp, but you can return to this chapter when you will have a little more understanding of Java. So, here it goes...

In Java there is a feature named `reflection`. Reflection can be used to inspect a package and access information of all its contents including private members. You can imagine that maybe this is not such a good thing, especially in a productive application that requires higher levels of security. Plus, such a feature makes it useless to have so many types of accessors, right? Up to Java 9, this is how things were in Java, and using reflection lead to problems included in the **Jar Hell** category. By introducing modules, **reflection can be restricted** as well. As in, reflection is no longer possible unless the module is configured to allow it. There are three forms of the same directive that can be used to configure access using reflection:

- `open` - is used at module declaration level, and allows reflective access to all packages within it

  ```
  open module chapter.zero {
     requires transitive org.apache.logging.log4j;

     exports com.apress.bgn.ch0 to chapter.three;
  }
  ```

- `opens` - is used inside the module declaration to selectively configure access through reflection only to certain packages

  ```
  module chapter.three {
      requires chapter.zero;

      opens com.apress.bgn.ch3.helloworld;
  }
  ```

- `opens ... to` - is used inside the module declaration to selectively configure access through reflection only to certain packages and to a specific module. It's a little difficult to give an example here, but let's imagine this project uses Spring Boot[5], and Spring Boot uses reflection to instantiate objects of types defined in our `chapter.three` module.

  ```
  module chapter.three {
      requires chapter.zero;
  ```

[4] More details here `https://logging.apache.org/log4j/2.x/`

[5] A very popular framework written in Java is Spring, Spring Boot is used to build production-ready Spring applications. More about it here:`https://projects.spring.io/spring-boot/`

```
        requires spring.boot;
        requires spring.web;
        requires spring.context;
        requires spring.boot.autoconfigure;
        opens com.apress.bgn.ch3.helloworld to spring.core;
    }
```

Aside from opening a package or a module for reflection the directives above also provide access to the package's (respectively all packages in the module) public types (and their nested public and protected types) at runtime only.

There are two directives left to cover: `uses` and `provides` that will be difficult to explain here, since they make more sense after you have dug into Java a little. But let's give it a try, and you can return to this chapter later. Module declarations that contain directives `provides` or `provides... with` are named `service providers` because **these modules provide a service implementation.**

Java has a class named `java.util.ServiceLoader` that can be used to modularize an application and load implementations of a service. What is a service in this case? A java object type, that defines only a contract, and that service providers need to provide a concrete implementation for.

! A service is modeled in Java using an object type that will be covered in **Chapter 4** called an `interface`. An interface can be compared with the earliest specification version for a car before being built, it covers what it should do, but not how. When the car is actually being built, that is the **implementation** part, that is when the *how* is decided. And cars do the same things, but in different ways depending on their type of engine, for example. So an electric car will have a different implementation (under the hood) for the braking function(during braking electric cars also recharge their battery) than a diesel based car.

Assuming the service needed to be implemented is named `NakedService`, a module that contains a class that provides an implementation for it, is declared as follows:

```
module chapter.one {
    requires chapter.zero;

    provides com.apress.bgn.ch0.service.NakedService
        with com.apress.bgn.ch1.service.Provider;
}
```

By using the `provides` declarative like that, this module just became a `service provider`.

The module that will use that service using the `java.util.ServiceLoader` class, is called a `service consumer` and to declare the fact that a service consumer will use an implementation of that service will contain the following directive in its module configuration file:

```
module chapter.three {
    requires chapter.zero;

    uses com.apress.bgn.ch0.service.NakedService;
}
```

You would ask now, but what is the difference from normally accessing module contents? Well, if you noticed, there is no `requires chapter.one` directive in the module configuration file depicted previously. Why? Because when using services it is only needed to have the module that provides the service type as a dependency in the

classpath. So, this way module `chapter.three` does not really care who provides the implementation, and as long as it is there is retrieved by the `java.util.ServiceLoader`. Why is this important? Because it decouples an application and removes explicit dependencies on concrete implementation(the `module-info.java` for module `chapter.three` does not need to explicitly declare a dependency on module `chapter.one`). This is the Java simple way to support **Inversion of Control**.

! It was mentioned in the initial chapter, that this book will introduce you into the core components of programming which are: data structures, algorithms, design patterns and most used coding principles. They will be explained as they come up in the book, so there is no predefined order or sequence that you should expect.
To explain **Inversion of Control**, the **Dependency Injection** must be explained. The action performed by any program (not only Java) is the result of interaction between its interdependent components, usually named objects. Dependency injection is a concept that describes how dependent objects are connected at runtime by an external party. Take a look at Figure A.4. It describes two types of relationships between objects, and how those objects "meet" each other.

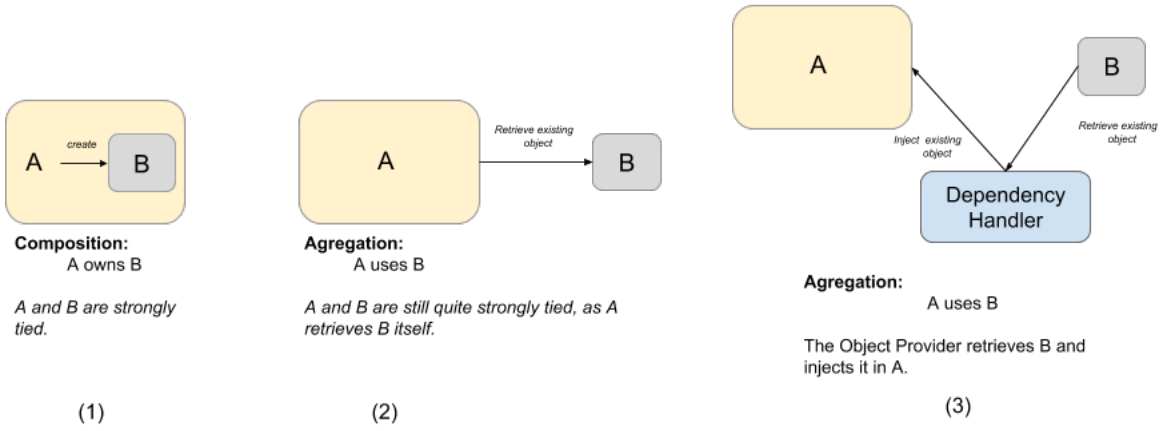

Figure A.4: Object relationships and how they meet.

So, object A that needs an object of type B to perform its functions, thus *A depends on B*. Object A can directly create the object B - case (1), `composition` or retrieves a reference to an existing object itself - case (2), `aggregation`, but this ties them up together.
Dependency injection allows severing that tie, by using an external party to provide an object of type B to the object of type A - case (3), still `aggregation`, but with no direct ties and a twist.
Inversion of Control is a design principle, in which generic reusable components are used to control the execution of problem-specific code, as in retrieving dependencies. Thus, you can say that the `java.util.ServiceLoader` is an dependency handler used to perform dependency injection and thus it was designed following the inversion of control principle.[6]

This is where the *under the hood* section ends. I hope it has given you enough understanding of Java organisation of code and reasons behind it, that you will continue reading the book and experimenting with code confidently.

[6]also known as the Hollywood Principle: "Don't call us, we'll call you".

A.2 Java Typical Application design

Let's consider a banking web application. It probably is quite complex, right? A lot of code has to be written to handle user requests, process banking transactions and saving and retrieving data. Can you imagine how this code is organized? Most applications have three basic layers: presentation, services and data access. Figure A.5 depicts this common structure:

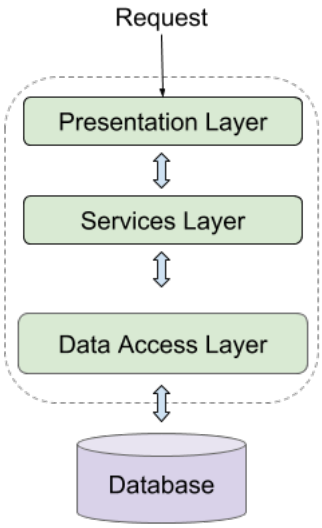

Figure A.5: The three most common layers of a Java application

In the most simple applications, each layer is represented by a sub-project or module, that contains all the code used to perform a group of related operations. For example: all code in the presentation layer implements functionalities for interaction with the user, processing user requests and transforming them into calls for layers under it. So before there were Java modules, java projects were themselves organized as multi-module projects. And that is why in IntelliJ IDEA there is a tab called `Modules`.

The "HelloWorld!" project compiled and executed manually

You've probably noticed the `Terminal` button in your IntelliJ IDEA. If you click that button, inside the editor a terminal will open. For Windows it will be a Command Prompt instance, for Linux and MacOs will be the default shell. And IntelliJ will open your terminal right into your project root. Here is what you have to do:

- enter the `src` directory by executing the following command:
  `cd src`
  `cd` is a command that works under Windows and Unix systems as well and is short for *change directory*
- compile the `HelloWorld.java` file by executing:
  `javac HelloWorld.java`
  `javac` is a JDK executable used to compile java files, that IntelliJ IDEA calls in the background as well
- run the resulting bytecode from the `HelloWorld.class` file by executing:
  `java HelloWorld`

Figure A.6 depicts the execution of those commands, in a terminal in IntelliJ IDEA.

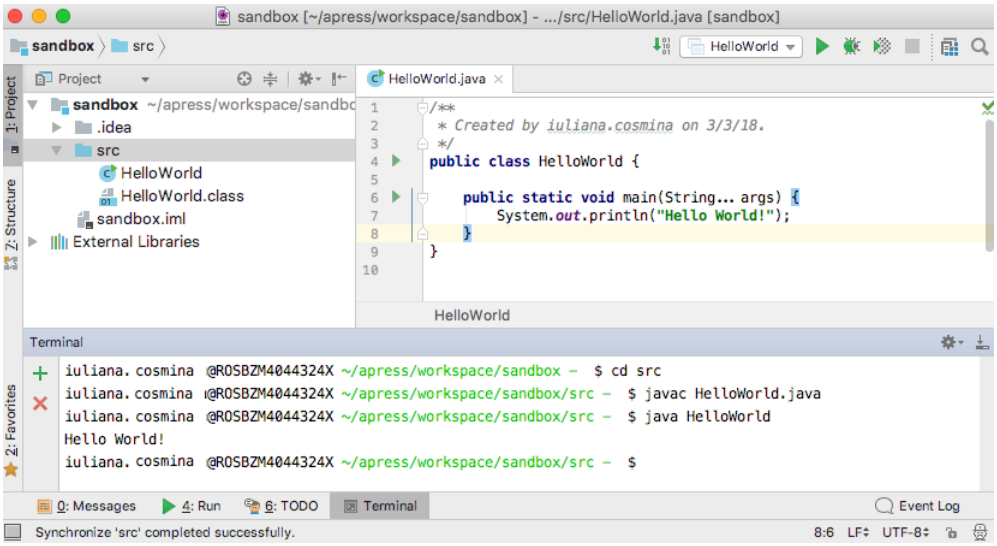

Figure A.6: Manually compile and run the HelloWorld class in a terminal inside IntelliJ IDEA

Looks simple right? And it actually is simple, because no packages or Java modules were defined. But wait, is that possible? Well, yes. If you did not define a package, the class is still part of an unnamed default package, that is provided by default by the JSE platform for development of small, temporary, educational applications, like the one you are just building. So let's make our project a little bit more complicated and add a named package for our class to be in.

Putting class "HelloWorld" in a package

In the creation menu there is a `Package` option. So click-right on the `src` directory and select it. A dialog window will appear where you have to enter the package name. Enter `com.sandbox`. In Figure A.7 that dialog windows is depicted, and even if the package is already created, we introduced the same name again, to show you how the IDE warns you that you are trying to create a package with the same name.

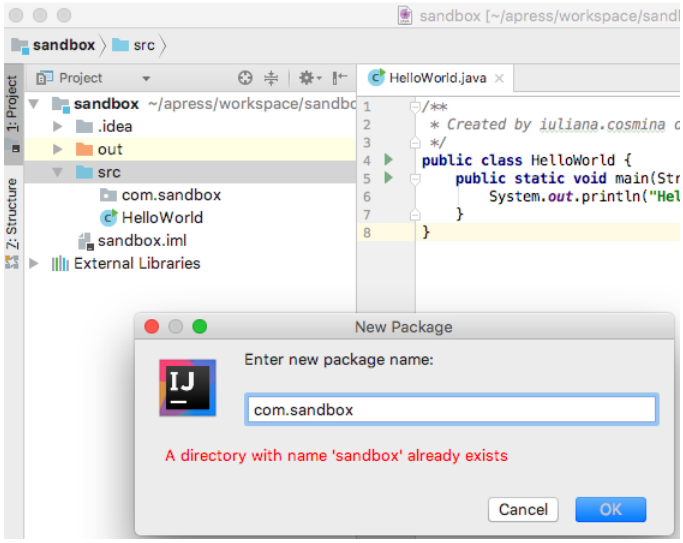

Figure A.7: Create package in IntelliJ IDEA

So, we created the package, but the class is not in it. Well, the way to get it there, is to select it and drag it into it. A dialog window for moving the class will appear, because the editor has to modify the class to make it to belong to the package by adding a `package` statement. And it requires your approval for the operation. The Figure A.8 depicts this dialog window.

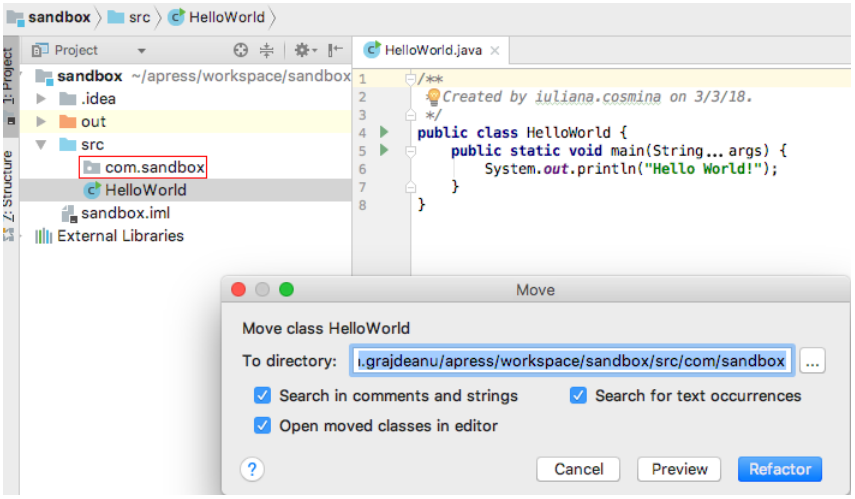

Figure A.8: Moving a class into a package in IntelliJ IDEA

Click on the `Refactor` button and look at what happens to the class. The class should now start with a `package com.sandbox;` declaration. If you rebuild your project, and then look at the directory structure, you will see something similar to what is depicted in Figure A.9

```
iuliana.cosmina @ROSBZM4044324X ~/apress/workspace/sandbox - $ tree
.
├── out
│   └── production
│       └── sandbox
│           └── com
│               └── sandbox
│                   └── HelloWorld.class
├── sandbox.iml
└── src
    └── com
        └── sandbox
            └── HelloWorld.java

8 directories, 3 files
```

Figure A.9: New directory structure after adding the com.sandbox package

Obviously if you compile and execute the class manually, you have to consider the package now, so your commands will change to:

```
~/sandbox/src -  $ javac com/sandbox/HelloWorld.java
~/sandbox/src -  $ java com/sandbox/HelloWorld
Hello World!
```

But things do not end here, because we still have Java modules. So, what about that? How is our code running without a `module-info.java` file? Well, there is a default unnamed module, and all JARs, modular or not, and classes on the classpath will be contained in it. This default and unnamed module exports all packages and reads all other modules. Because it does not have a name it cannot be required and read by named application modules. Thus, even if your small project seems to work with JDKs with versions bigger or equal to 9, it cannot be accessed by other

modules, but it works because it can access others. (This ensures backwards compatibility with older versions of the JDK) This being said, let's add a module in our project as well.

Configuring the "com.sandbox" module

Configuring a module is as easy as adding a `module-info.java` under the `src` directory. In Figure A.6, in the menu listed there there is a `module-info.java` option and if you select that, the IDE will generate the file for you. All is well and fine, and if you do not like the module name that was generated for you, you can change it. I changed it to `com.sandbox` to respect the module naming convention mentioned in the modules section.

```
/**
 * Created on 3/3/18.
*/
module com.sandbox {

}
```

What happens now that we have a module? Not much from the IDEs point of view. But if you want to compile a module manually you have to know a few things. I compiled our module using the following command:

```
~/sandbox/src/ - $ javac -d ../out/com.sandbox \
        module-info.java  \
        com/sandbox/HelloWorld.java
```

! "\" is a MacOS/Linux separator. On Windows either write the whole command on a single line or replace "\" with "^".

Let me explain what I did there. The syntax to compile a module is this:

```
javac -d [destination location]/[module name]    \
    [source location]/module-info.java \
    [java files...]
```

The result of executing that command, will be that under the `out` directory, a directory named `com.sandbox` will be created - the module name. Under this directory, we'll have the normal structure of the `com.sandbox` package. The contents of the `out` directory are depicted in Figure A.10

Figure A.10: Java module com.sandbox compiled manually

As you have noticed in this example, the module does not really exist until we compile the sources, because a java

module is more of a logical mode of encapsulating packages described by the `module-info.class` descriptor. The only reason the `com.sandbox` directory was created is that we specified it as argument in the `javac -d` command.

We have a compiled module, what do we do with it? We try to run the application obviously.

```
sandbox/ - $ java --module-path out \
      --module com.sandbox/com.sandbox.HelloWorld
Hello World!
```

The syntax to execute a modular application is this:

```
java --module-path [destination location]  \
   --module [module name] /[package name].HelloWorld
Hello World!
```

Regarding the module name, doesn't it seem a little redundant? To me it sure looks like it, that is why I prefer not to create directories for modules unless I have more of them under directory `src`. And we also have to talk about the standard naming conventions for modules. That is also another thing that might give developer headaches if they want to create directories for modules. In multiple blog articles and even in the Oracle Magazine from September 2017, this is recommended.[7] But do not worry about it for now, the sources of the book contain modules with simple names and the module configuration is already in place for you.

Let's give it a try. Let's create two modules, one named `com.sandbox.one` and one named `com.sandbox.two`, create directories for their sources too. In Figure A.11 you can see how the project structure looks in this case. You can see that even the IDE is puzzled about it, because the project view has changed into something that does not resemble what was depicted in the previous figures.

[7] Oracle Magazine edition from September 2017 can be accessed here: `http://www.javamagazine.mozaicreader.com/SeptOct2017#&pageSet=29&page=0`

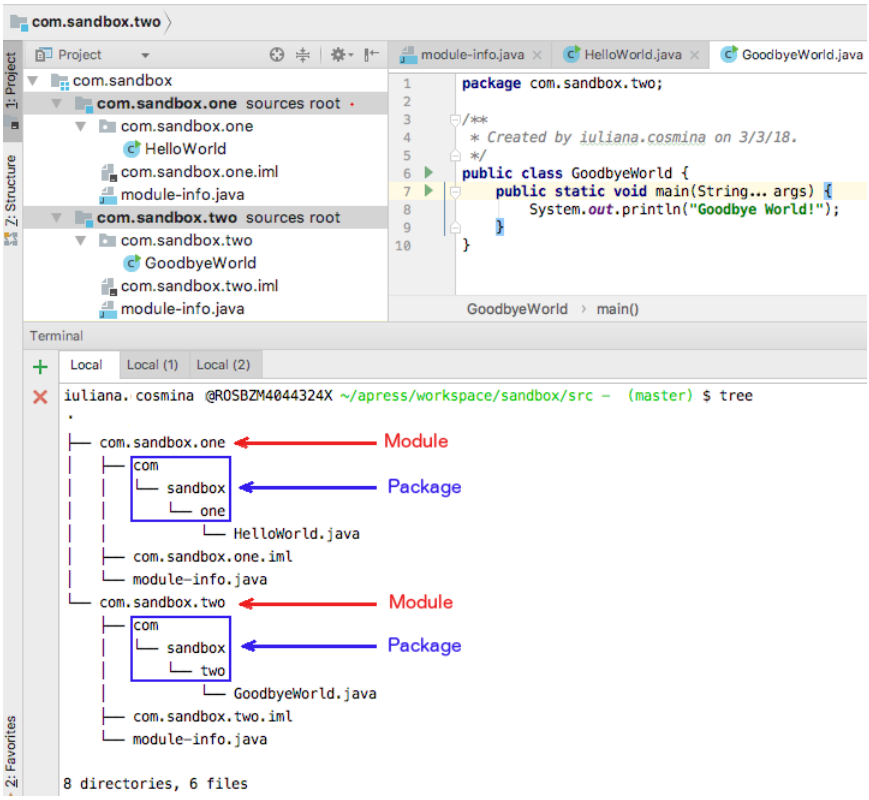

Figure A.11: Java modules with matching directory names

I cannot recommend you what approach to use, because there might be developers out there for whom this makes sense. I consider it redundant and I will try to avoid having more than one java module in the same project module.

As you can have figured out so far, compiling and executing Java applications manually is no walk in the park. That is why this function can be fulfilled by smart Java editors, like IntelliJ IDEA, Eclipse or Apache Netbeans and with build tools. Build tools were introduced to generate those kind of statements for you, to automatically add dependencies to the classpath/ modulepath and overall make development quicker and easier. We did not cover compiling and executing applications that depend on external JARs/modules because that is not the scope of this book. But if you are really interested in this, the internet is great and helpful. And now that we made it clear that manual compiling and execution of Java applications is a pretty much a no-go (it's possible, but not worth the effort), let's start talking about Java build tools.

Java projects using build tools, mostly Gradle

Maven is a build automation tool used primarily for Java projects. Although Gradle is gaining ground, Maven is still one of the most used build tools. Tools like Gradle and Maven are used to organise the source code of an application in interdependent project modules and configure a way to compile, validate, generate sources, test and generate artifacts automatically. An artifact is a file, usually a JAR, that gets deployed to a Maven repository. A Maven repository is a location on a HDD where JARs get saved in a special directory structure.

The discussion about build tools must start with Maven[8], because this build tool standardized a lot of the terms

[8]We won't mention Ant, which is the first Java build tool, that is still hanging around because of legacy code. More about it here: `https://ant.apache.org/`

we used in development today. For example: for big projects, that can be split into multiple parts - `sub-projects` - that can be compiled and archived in separate JARs, we call a sub-project a `module`, because the XML element to declare a sub-project as a child of a project is called `<module ...>`.

A Maven build produces one or more artifacts, such as a compiled JAR, a "sources" JAR and even "test" jar if the module has test classes in it. Maven introduced a way to uniquely identify artifacts. Each artifact has a `group ID` (usually a reversed domain name, like `com.sandbox`), an `artifact ID` (just a name), and a version string. Maven projects are configured using one or more `pom.xml` files, one for the parent project and one or each of its modules(sub-projects/children/modules). In the following code snippet, you can see a simplified version of a parent project `pom.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    <groupId>com.sample</groupId>
    <artifactId>parent-maven</artifactId>
    <version>1.0-SNAPSHOT</version>

    <packaging>pom</packaging>

    <modules>
        <module>child-01</module>
        <module>child-02</module>
    </modules>
    ...
</project>
```

This file has the minimum elements needed to understand how the project is organized. You can see that the project is identified uniquely by the values of elements: `<groupId>` , `<artifactId>` and `<version>`. And under them, you can see what are its sub-projects(modules). Look at Figure A.12. Is that the same structure you imagined after seeing that file?

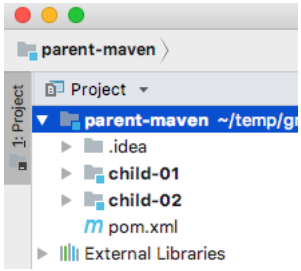

Figure A.12: Maven multi-module project structure

Maven also requires the project structure to look a certain way, it introduced a standard for this as well that has taken over the Java development world. In Figure A.13, you can see the structure of the `parent-maven` project and one of its modules `child-01`.

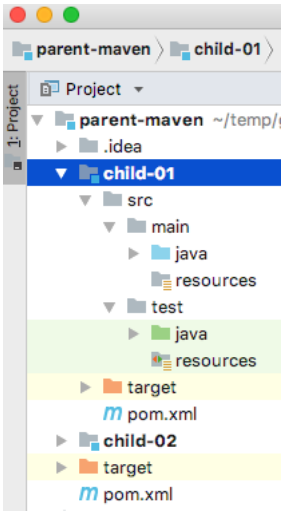

Figure A.13: Maven module structure

Now, let's explain! Project `parent-maven` is called a `multi-module` project because it is made up of more than one module(`child-01` and `child-02` here). The modules are in fact sub-projects, that are compiled and archived into JARs. Each module is represented by a directory containing the sources, resources, tests and a Maven configuration file. The directories are located under the `parent-maven` directory. It's not a must for them to be there, but it definitely makes it obvious that they are parts of a bigger project.
Sub-Project `child-01` is a module of project `parent-maven`. The content of this directory gets build into an artifact, thus it definitely contains Java code in there. Its internal structure can be explained as follows:

- the `src` directory contains the sources of this module that are grouped in two categories stored under two directories: `main` and `test`
- the `main` directory contains source and resource files that get built into the artifact
  - the `java` directory contains packages declarations, with their classes and `module-info.java` files
  - the `resources` directory contains any type of text files(*.properties, *.XML, *.JSON, etc) that are read by classes declared into the `java` directory(configuration and data)
- the `test` directory has an identical internal structure as `main`, only the classes in its `java` directory are java test classes, used to test the classes defined in `main\java` and the `resources` directory contains files on any text type filled with test data and test configurations. This module can be build into an artifact as well and its name will have `*-tests` as a suffix. Such artifacts are used as dependencies by other modules that contain tests that use the API defined in their test dependency.

The Maven way to identify artifacts and the internal structure of projects and multi-module projects have become the standard for Java applications. Gradle, the next best thing when it comes build tools for Java applications, has embraced them both. Gradle is preferred nowadays for a number of reasons, but the one you will hear being mentioned the most, is the fact that configuration files are written in Groovy and the configuration for big projects can be reduced a lot. (Maven uses XML with strict rules for configuration, and for big projects the configuration files tend to become quite complex and unreadable).[9]

For example, to declare a parent project with two modules, like the one listed previously for Maven, using Gradle you just need a `settings.gradle` file containing the following:

[9]The Maven project was presented as a case-study here is a real and working project that can be found on GitHub here: `https://github.com/iuliana/gradle-vs-maven`. If you are curious you can take a look.

```
rootProject.name = 'parent-gradle'

include 'child-01'
include 'child-02'
```

And a `build.gradle` file containing:

```
allprojects {
    version = '1.0-SNAPSHOT'
}
```

The `settings.gradle` and the `build.gradle` file are both Groovy scripts. Only one `settings.gradle` script will be executed in each build (in comparison to the `build.gradle` script in multi-project builds), that is why this is the proper place in which the project structure is defined. You can see that even with two configuration files, the syntax is still greatly reduced and easier to read.

Gradle was chosen as the go-to build tool for the sources attached to this book. A multi-module project can be downloaded from GitHub, and build in the command line or imported into IntelliJ. This approach will make sure that you get quality sources, that can be compiled in one-go. It is also practical, because I imagine you do not want to load a new project in IntelliJ every time you start reading a new chapter. Also, it makes it easier for me to maintain the sources and adapt them to a new JDK, and with Oracle releasing so often, I need to be able to do this quickly.

The project you will use to test the code written in this book and write your own code if you want to, is called `java-bgn`. It is a multi-module Gradle project. The first level of the project is the `java-bgn` project, that has a configuration file named `build.gradle`. In this file, all dependencies and their versions are listed. The child projects, the ones on the second level, are the modules of this project. And we call them *"child"* projects because they inherit those dependencies and modules from the parent project. In their configuration files, we can specify what dependencies are needed, from the list defined in the parent. And these modules are actually a method of wrapping up together sources for each chapter and that is why these modules are named `chapter00`, `chapter01`, etc. If a project is big and needs a lot of code to be written, the code is split again in another level of modules. Module `chapter05` is such a case, and is configured as a parent for the projects underneath it. In Figure A.14 you can see how this project looks like loaded in IntelliJ IDEA, and module `chapter05` is expanded so you can see the third level of modules. Each level is marked with the corresponding number.

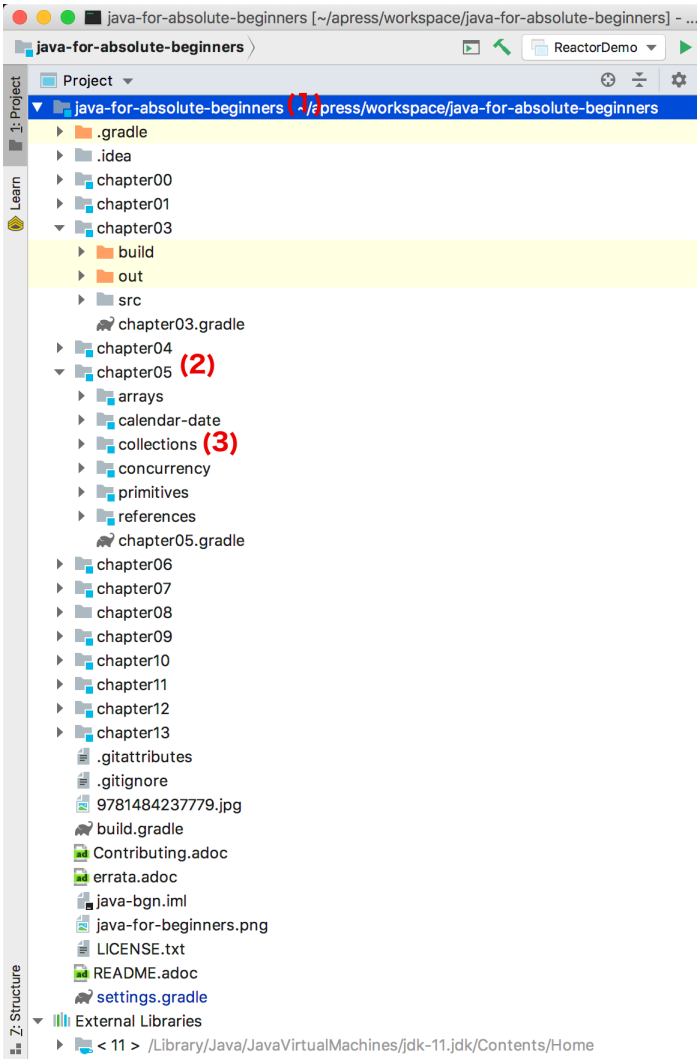

Figure A.14: Gradle multi-module level structure

If you have loaded it into IntelliJ IDEA like you were taught in **Chapter 2**, you can make sure everything is working correctly by building it. Here's how you do it.
You can do it by using the IntelliJ IDEA editor, in the upper right side you should have a tab called `Gradle projects`. If the projects are loaded like they are depicted in Figure A.15, the project was loaded correctly. If the `Gradle projects` tab is not visible just look for a label like the one marked with (1) and click on it. Expand the `java-bgn(root)` node until you find the `build` task, marked with (2). If you double click it and in the view at the bottom of the editor you do not see any error, all your projects were build successfully.

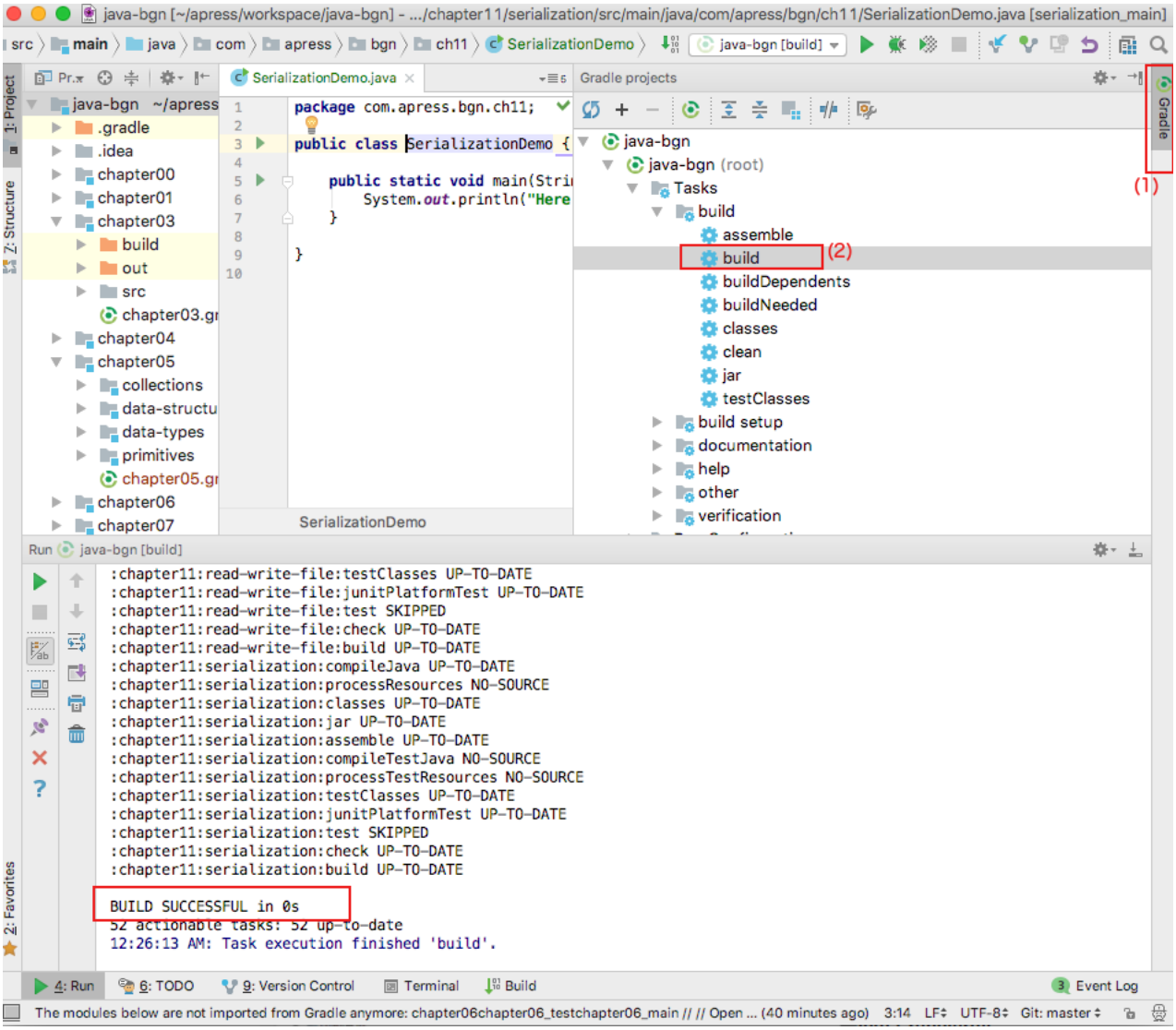

Figure A.15: Gradle multi-module level structure

The second way to make sure the Gradle project is working as expected is to build it from the command line. Open an IntelliJ IDEA terminal, and if you installed Gradle on the system path as explained in **Chapter 2** just type `gradle clean build` and hit `<Enter>`. In the command line, you might see some warnings, if the Gradle plugin for supporting Java modules is still unstable when this book reaches you, but as long as the execution ends with `BUILD SUCCESSFUL` everything is all-right.

Aside from the `sandbox` project, all the classes, modules and packages mentioned in this section are part of this project. The `chapter00` and `chapter01` do not really contain classes specific to those chapters, I just needed them to be able to construct the java module examples. IntelliJ IDEA sorts modules in alphabetical order, so the naming for the chapter modules was chosen this way so they are listed in the normal oder you should work with them.