9a

Modular Programming in Java

Objectives

- The benefits of modular programming.
- Declaration of a Java Module.
- Module descriptor.
- Module directives.
- Create and run a Modular project with IntelliJ.
- The unnamed module.
- Create and run unnamed module.
- Add a Modular JAR and use it with IntelliJ

6a.1 Problems in Java application

Problems in developing or delivering applications with Java SE 8 or earlier systems:

- As **JDK is too big**, it is rather **difficult to scale it down** to small devices. This problem persists although Java SE 8 has introduced 3 types of compact profiles to solve this problem (compact1, compact2, and compact3).
- JAR files like rt.jar and other **standard packages are too big** to use in small devices and applications.
- As JDK is too big, our applications or devices are **not able to support better Performance**.
- Java **applications become too big** as well.
- Everyone can access internal non-critical APIs because public access is too open. Therefore, **Security is big issue**.
- There is **no Strong Encapsulation** in the current Java System because "public" access modifier is too open. As JDK, JRE is too big, it is hard to Test and Maintain applications.
- It is difficult to support **Less Coupling** between components.

6a.1 Advantages of using Java modules

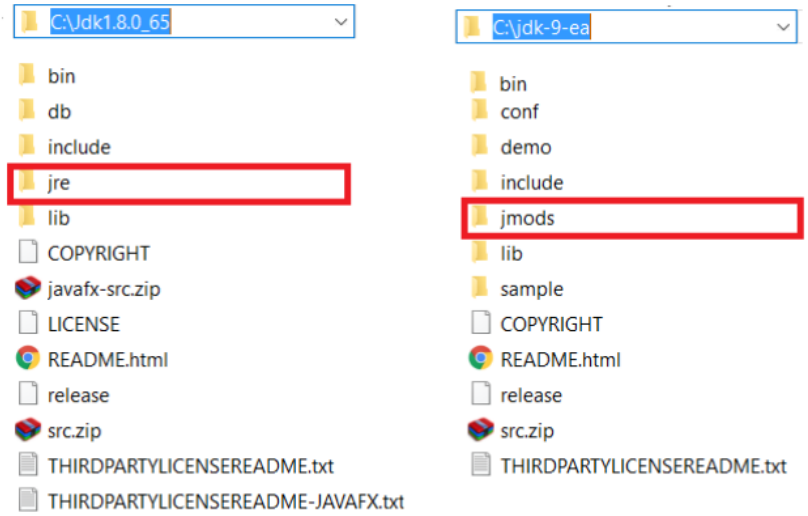
Java SE 9 Module System offers the following benefits:

- Java SE 9 and later editions divide JDK, JRE, JARs etc, into smaller modules. Thus, we can use whatever modules we want. This way Java **Applications scale down easily** to Small devices.
- **Enhances Testing and Maintainability**.
- Provides **better Performance**.
- **Supports very Strong Encapsulation** once public access is not just public.
- Does **not allow direct access to Internal Non-Critical APIs** anymore.
- **Provides better Security** once Java Modules hide unwanted and internal details in a safe way.
- Java **applications become smaller** because we can use only what ever modules we want.
- Its easy to **support Less Coupling** between components.
- Its easy to **support Single Responsibility Principle** (SRP).

6a.1 Advantages of using Java modules

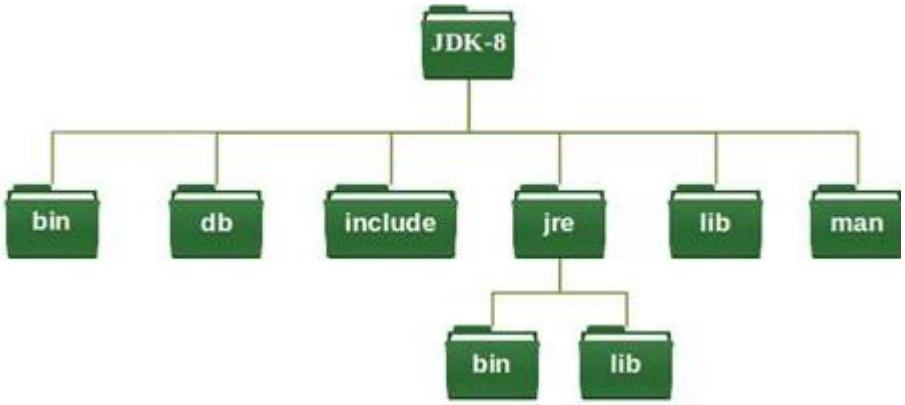
Features of Java SE 9 Module System:

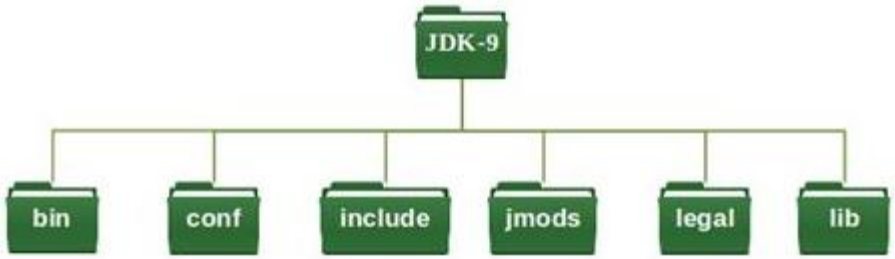
- **Scalable Java platform**—Previously, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain and evolve. It couldn't be easily subsetted. The platform is now modularized into 95 modules (this number might change as Java evolves). **You can create custom runtimes consisting of only modules you need for your apps or the devices you're targeting.** For example, if a device does not support GUIs, you could create a runtime that does not include the GUI modules, significantly reducing the runtime's size.

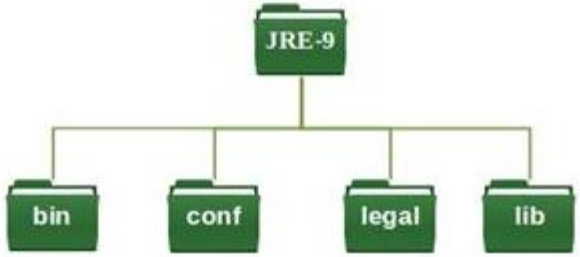


6a.1 Advantages of using Java modules

- **Reliable configuration**—Modularity provides mechanisms for **explicitly declaring dependencies** between modules in a manner that's recognized both at compile time and execution time. The system can walk through these dependencies to **determine the subset of all modules required to support your app.** The directory structure of JDK and JRE is almost the same, except that JDK has two additional directories: **jmods** and **include**. Also, there is **no JRE sub directory** in JDK9 and later editions.







6a.1 Advantages of using Java modules

- **Improved performance**—The JVM uses various optimization techniques to improve application performance.
(Java Specification Requests) **JSR 376** indicates that these techniques are more effective when it's known in advance that required types are located only in specific modules.
- **Strong encapsulation**—The packages in a module are accessible to other modules only if the module explicitly exports them. Even then, another module cannot use those packages unless it explicitly states that it requires the other module's capabilities. This improves platform security because fewer classes are accessible to potential attackers. You may find that considering modularity helps you come up with cleaner, more logical designs.
- **Greater platform integrity**—Before Java 9, it was possible to use many classes in the platform that were not meant for use by an app's classes. With strong encapsulation, these internal APIs are truly encapsulated and hidden from apps using the platform. This can make migrating legacy code to modularized Java 9 problematic if your code depends on internal APIs.

6a.1 Advantages of using Java modules

"*A key motivation of the module system is strong encapsulation*"

By default, a type in a module is not accessible to other modules unless it's a **public type and** you **export its package.** You expose only the packages you want to expose .

Types of modules:

- **Standard modules** – implement the Java Language SE Specification (package names starting with **java**),
- **JavaFX modules** – (package names starting with **javafx**),
- **JDK-specific modules** (package names starting with **jdk**) and
- **Oracle-specific modules** (package names starting with **oracle**).
- **User- defined** (package names are unique user- defined).

Module names may be followed by a version string.

For example, @9 indicates that the module belongs to Java 9.

6a.1 Advantages of using Java modules

Note:

Java Module System has a "`java.base`" module.

It's known as the base Module. **It's an Independent module and does NOT dependent on any other modules**. By default, all other Modules dependent on this module. It's default module for all JDK Modules and User-Defined Modules.

6a.2 Java Module Declarations

A module is a named, self-describing collection of code and data. Its code is organized as a set of packages containing types, i.e., Java classes and interfaces; its data includes resources and other kinds of static information.

Mark Reinhold (Chief architect of the Java Platform Group at Oracle)

A **Java *module*** is a **uniquely named, reusable group of related packages**, as well as **resources** (such as images and XML files) and a **module *descriptor*** specifying:

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

6a.3 Java Module Declarations

A **Java module** must provide **a module descriptor– metadata that describes the module's dependencies, the packages the module makes available to other modules, and more.**

A **module descriptor** is the compiled version of a module declaration that's defined in a file named **`module-info.java`**. Each **module declaration** begins with the **keyword `module`**, followed by a **unique *`modulename`*** and a **module body** enclosed in braces, as in:

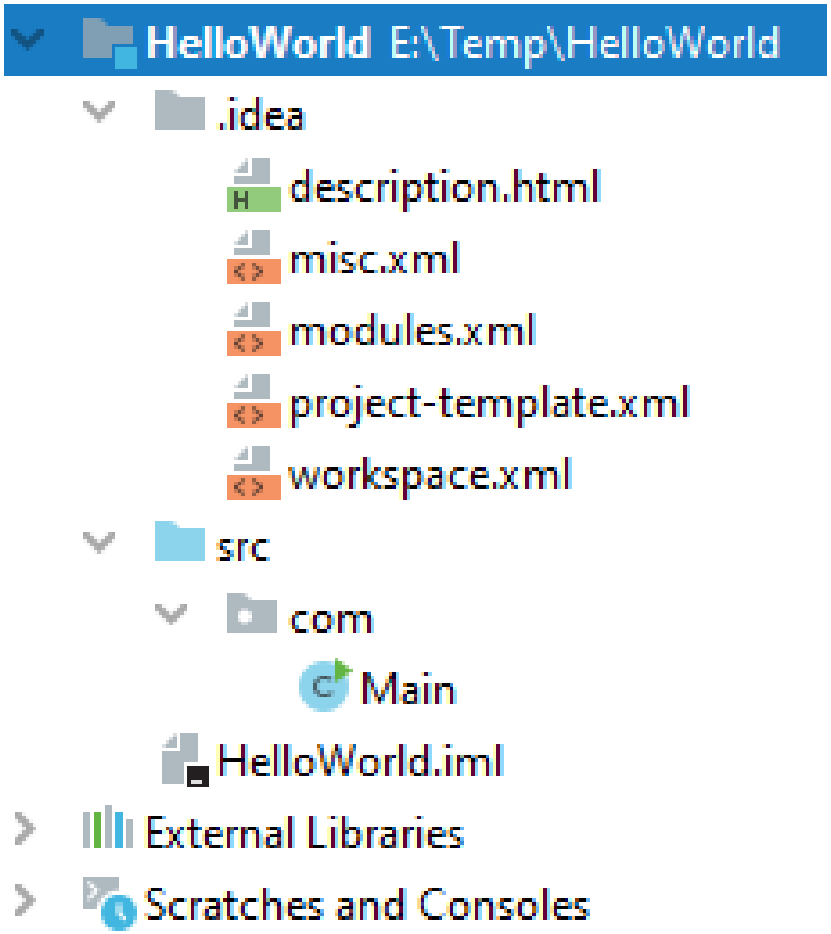
```
module modulename {
// module body
}
```

The module declaration's **body can be empty** or **may contain various module directives**, including **`requires`**, **`exports`**, **`provides`**…**`with`**, **`uses`** and **`opens`**. Compiling the module declaration creates a file named **`module-info.class`** in the module's root folder.

6a.4 IntelliJ modules VS Java modules

In **IntelliJ IDEA**, a **module** is an essential part of any **project**. An **IntelliJ** module is a discrete unit of functionality which you can compile, run, test and debug independently. It is created automatically together with a **project.**
Projects can contain multiple **modules.** You can add new **modules**, group them, and unload the **modules** you don't need at the moment. **Modules** consist of a **content root** and a **module** file. **Configuration information** for a module is stored in a **.iml module file**. By default, such a file is located in the module's **content root folder**.

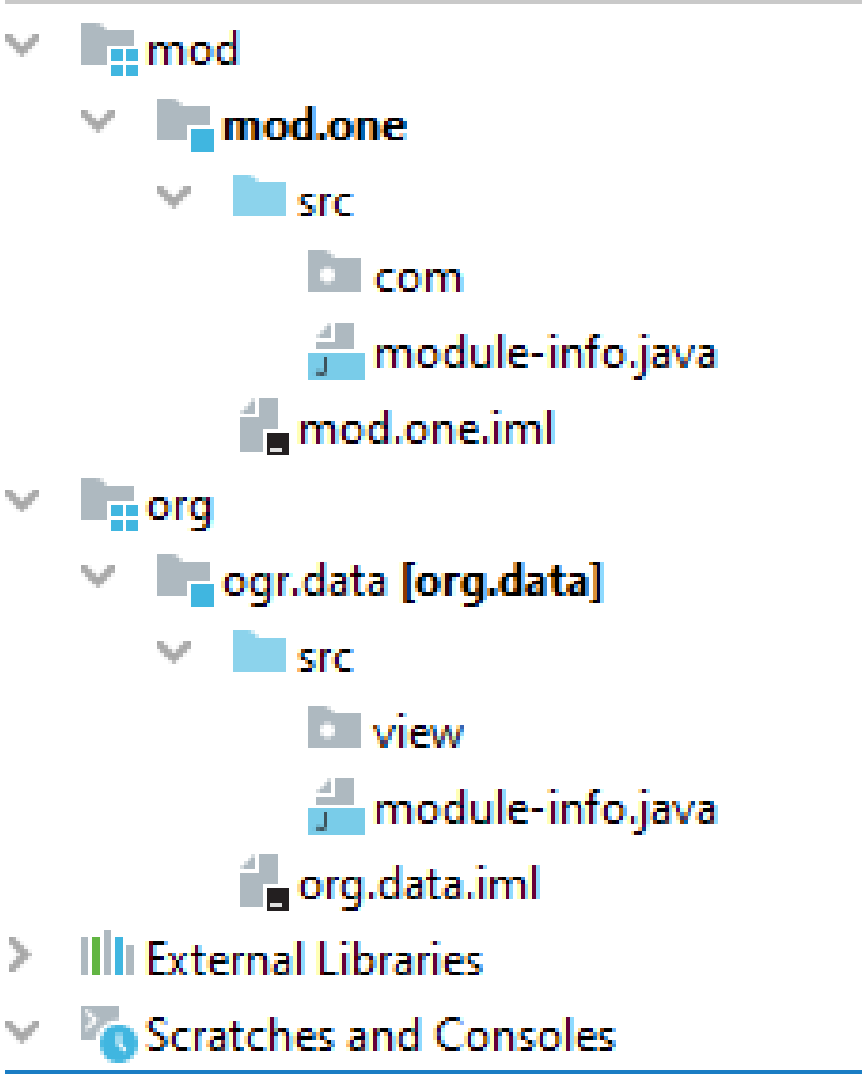
6a.4 IntelliJ modules VS Java modules



Content roots in IntelliJ IDEA are shown as or

A Content root contains folder of several categories. One important folder inside the Content root is the **Source root** denoted as

6a.4 IntelliJ modules VS Java modules



Java 9 modules map to **IntelliJ IDEA modules and provide additional features** via the **module descriptor** specifying:

- 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**

6a.4 IntelliJ modules VS Java modules

Summary:

IntelliJ IDEA allows to create a group of IntelliJ IDEA modules 

Every IntelliJ IDEA module  builds its own **classpath**.

With the introduction of the new **Java platform module system**, IntelliJ IDEA modules  can extend their capability by supporting the Java platform's **module-path** if it is used instead of the **classpath**.

Java modules are a way to strongly encapsulate your classes. It does not provide any means to control the version of an artifact you are using.

6a.4 IntelliJ modules VS Java modules

Summary:

It's important to note that there are two systems of modularly here. Firstly, the IntelliJ IDEA modules and secondly, the Java 9 modules that are configured using **module-info.java**. To use Java 9 modularity, each Java module needs to correspond to an IntelliJ IDEA module.

Also note that in IntelliJ IDEA 2017.1 and later you need to declare your IntelliJ IDEA **module dependencies** (**Project Structure-> Dependencies**) as well as your Java 9 module dependencies (**module-info.java**)

6a.3a Module Directives

The module descriptor `module-info.java` may contain the following directives:

`requires`

A **`requires`** module directive specifies that **this module depends on another module.**

This relationship is called a module dependency.

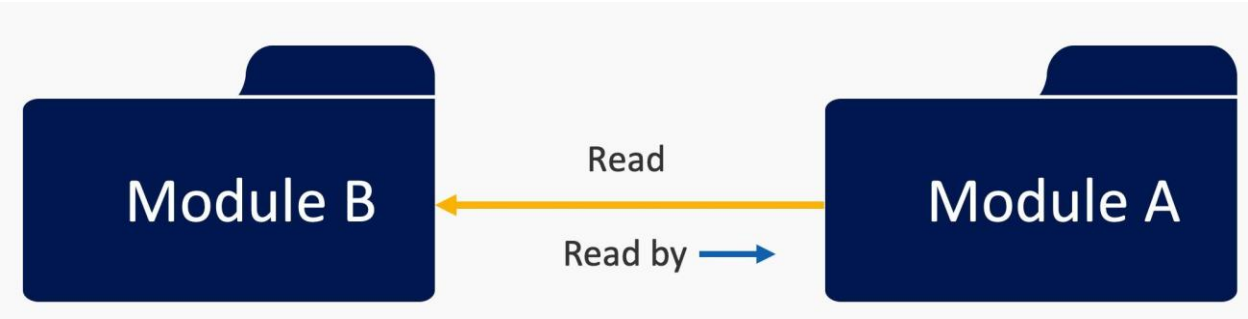
Each module must explicitly state its dependencies. When **module A requires module B, module A is said to read module B and module B is read by module A.**

To **specify a dependency on another module**, use requires, as in:

```
module usr.module{
 requires abc.xyz;  //abc.xyz is a module name
}
```

There is also a **`requires static`** directive to indicate that a module is **required at compile time** but is **optional at runtime.** This is known as an **optional dependency .**

6a.3a Module Directives



When module A requires module B, module A is said to **read** module B and module B is **read by** module A.

6a.3a Module Directives

`exports and exports…to.`

An **`exports`** module directive **specifies one of the module's packages** whose **`public` types** (and their nested **`public`** and **`protected`** types) **should be accessible to code in all other modules**.

```
module abc.xyz{

  exports com.foo.block; //com.foo.block represents a package

 }
```

It is important to understand that even if classes in a given package are **`public`**, they cannot be visible outside (both at compile time and runtime) if their packages are not explicitly exported via **`exports`** directive. Modules, therefore, can hide **`public`** functionality and that improves Java encapsulation system.

`exports…to`

An **`exports…to`** directive enables you to **specify in a comma-separated list** precisely **which module's or modules' code can access** the exported package. This is known as a *qualified* export.

6a.3a Module Directives

`uses.`

A **`uses`** module directive **specifies a service** used by this module, making the module **a service consumer**. A ***service*** is an **object** of a **class** that **implements the `interface`** or extends the **`abstract class` specified in the `uses`** directive.

`provides…with.`

A **`provides…with`** module directive specifies that **a module supplies a service implementation**, thus **making the module a service provider.**

The **`provides` part** of the directive **specifies** an **`interface or abstract class`** **listed** in a module's **`uses` directive** and the **`with` part** of the directive **specifies** the **name of the service provider class** that implements the **`interface`** or **`extends`** the **`abstract class`**.

6a.3a Module Directives

A Java module that wants to implement a service interface from a service **`interface`** module must:

- ✓ Require the service **`interface`** module in its own module descriptor.
- ✓ Implement the service **`interface`** with a Java **`class`**.
- ✓ Declare the service **`interface`** implementation in its module descriptor.

Imagine that the **`com.control.myservice`** **module** contains an **`interface`** named **`com.control.myservice.MyService`** . Imagine too, that you want to **create a service implementation** module for this service interface. Imagine that your implementation is called **`com.code.myservice.MyServiceImpl`**.

To declare that service implementation the module descriptor for the service implementation module would have to look like this:

```
module com.code.myservice {
   requires com.control.myservice;
   provides com.control.myservice.MyService with
            com.code.myservice.MyServiceImpl
}
```

The module descriptor **first declares** that it **requires** the service **`interface`** module. **Second**, the module descriptor **declares** that it **provides an implementation** for the service interface **`com.control.myservice.MyService`** via the class **`com.code.myservice.MyServiceImpl`** .

6a.3a Module Directives

`open, opens, and opens…to.`

Before Java 9, reflection could be used to learn about all types in a package and all members of a type, even its **`private`** members, whether you wanted to allow this capability or not. Thus, nothing was truly encapsulated.

`opens` `<package name>`

indicates that a **specific package's `public`** types (and their nested **`public`** and **`protected`** types) are **accessible to code in other modules at runtime only**. Also, all the types **in the specified package** (and all of the types' members) are accessible via reflection.

An **`opens…to` module directive** of the form

`opens package to` `<list of modules>`

indicates that a specific package's **`public`** types (and their nested **`public`** and **`protected`** types) are accessible to code in the listed modules **at runtime only**. All of the types in the specified package (and all of the types' members) are accessible via reflection to code in the specified modules.

6a.3a Module Directives

Specifying a name for the module is mandatory. Two modules within the same code base can't have the same name. It's good practice to name our modules the same way we name packages: **by using the domain names in reverse order**.

The name of the module could therefore be a prefix of the names of its exported packages, but we can name our modules how we want because we don't have any constraints regarding the format of the module name. Nevertheless, the name of the module complies with the general rules of identifiers in Java. A module can have the same name as a Java class or an interface, because the names of the modules have their own namespace

6a.3a Module Directives

Recommendations for module naming:

- Module names must be **reverse-DNS**, just like package names, e.g. **org.company.time**.
- Modules are a group of packages. As such, the **module name must be related to the package names**.
- Module names are **strongly recommended** to be the **same as the name of the super-package**.
- Creating a module with a particular name **takes ownership** of that package name and everything beneath it.
- As the owner of that namespace, any sub-packages may be grouped into sub-modules as desired so long as no package is in two modules.

6a.3a Module Directives

Thus the following is a well-named module:

```
module org.company.time {
   requires org.company.convert;
   // super package
   exports org.company.time;
   exports org.company.time.chrono;
   exports org.company.time.format;

}
```

6a.3a Module Directives

As can be seen, the m**odule contains a set of packages** (**exported** and **hidden**), **all under one super-package having the same name as the module.**

The **module name is the same as the super-package name (both names are identical but specify different entities- module and package**). The author of the module is asserting control over all names below **org.company.time** and could create a module **org.company**.time.**sometime** in the future if desired

6a.3a Module Directives

Cyclic dependencies aren't allowed by the module system at compile-time

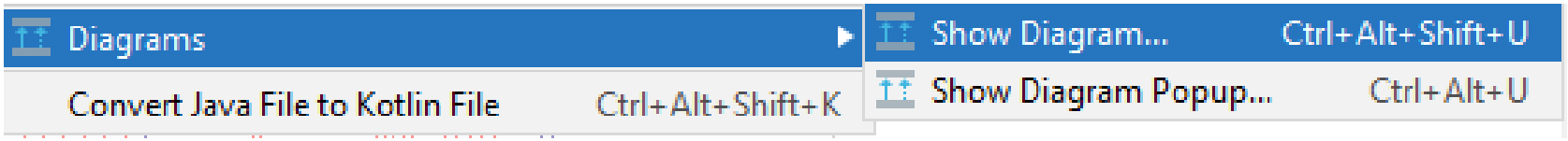
```
// module-info.java (module com.ap.moduleA)
module com.ap.moduleA {
    requires com.ap.moduleB;
}
// module-info.java (module com.ap.moduleB)
module com.ap.moduleB {
   requires com.ap.moduleC;
}
// module-info.java (module com.ap.moduleC)
module com.ap.moduleC {
    requires com.ap.moduleA;
}
```

6a.3a Module Directives

View module dependencies diagram

1. In the Project tool window, select an item (project/module) for which you want to view a diagram.

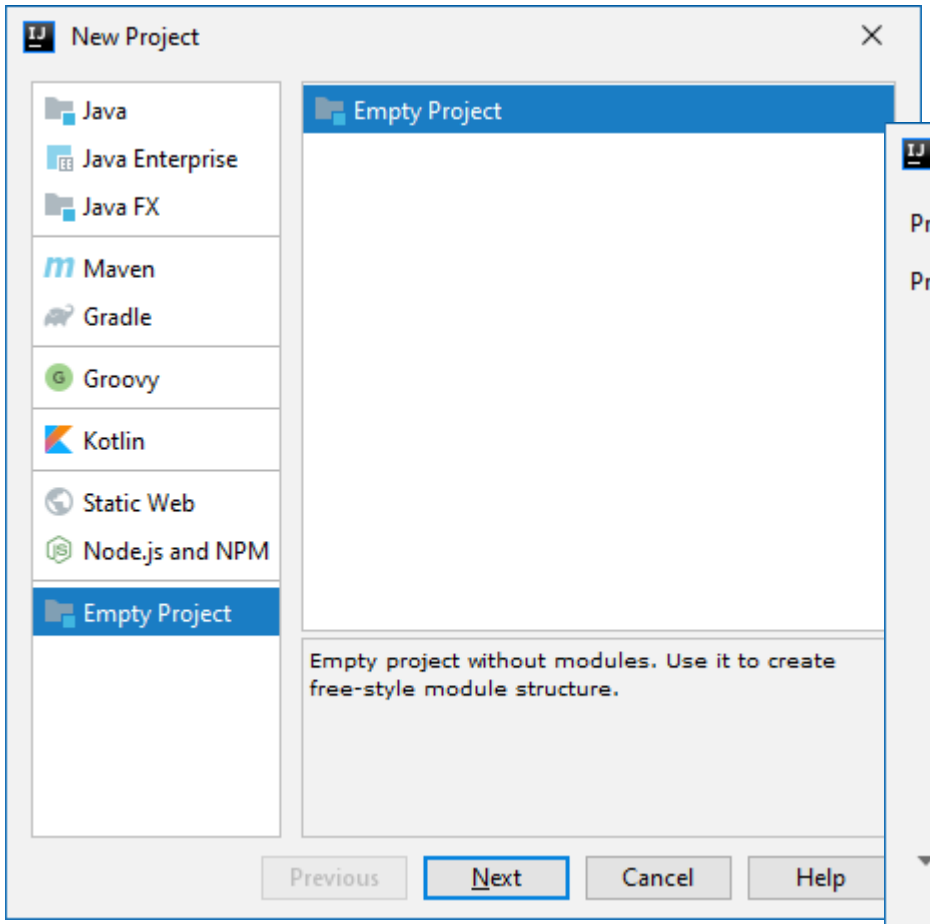
2. Right-click the selected item and from the context menu, select **Diagram | Show Diagram** **Ctrl+Shift+Alt+U**.

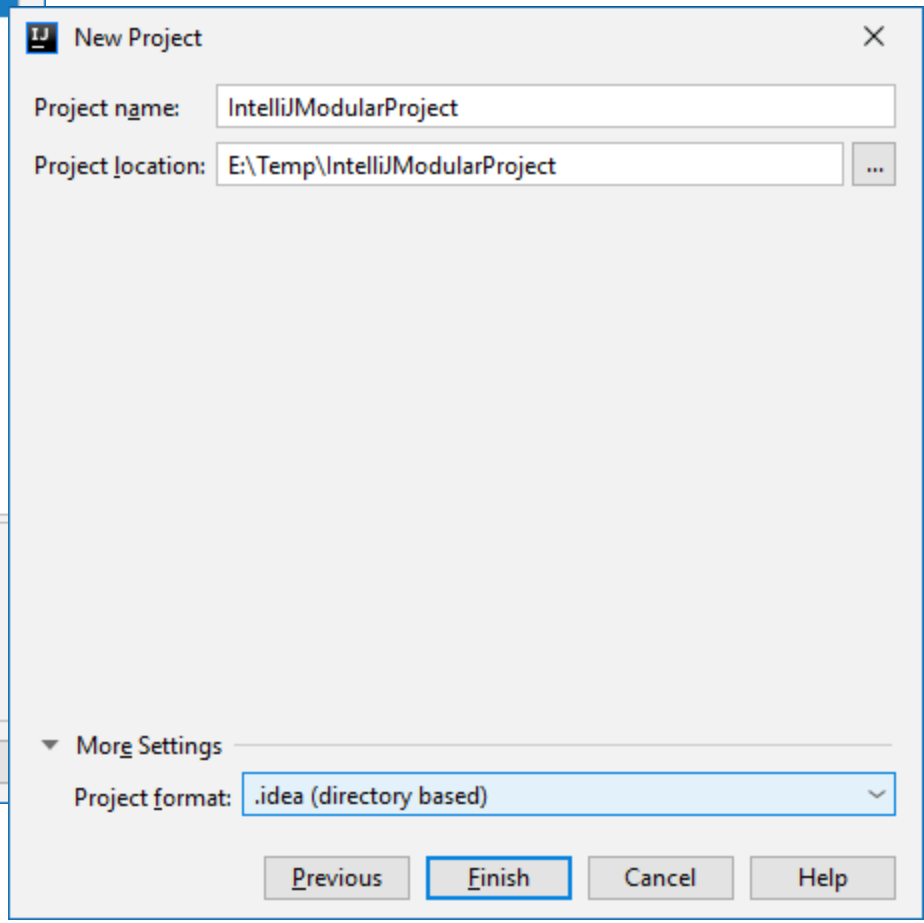


3. From the list that opens, select a type of the diagram you want to create.

6a.4 Modular Programming with IJ

1. Create a New Empty Project with IntelliJ IDEA





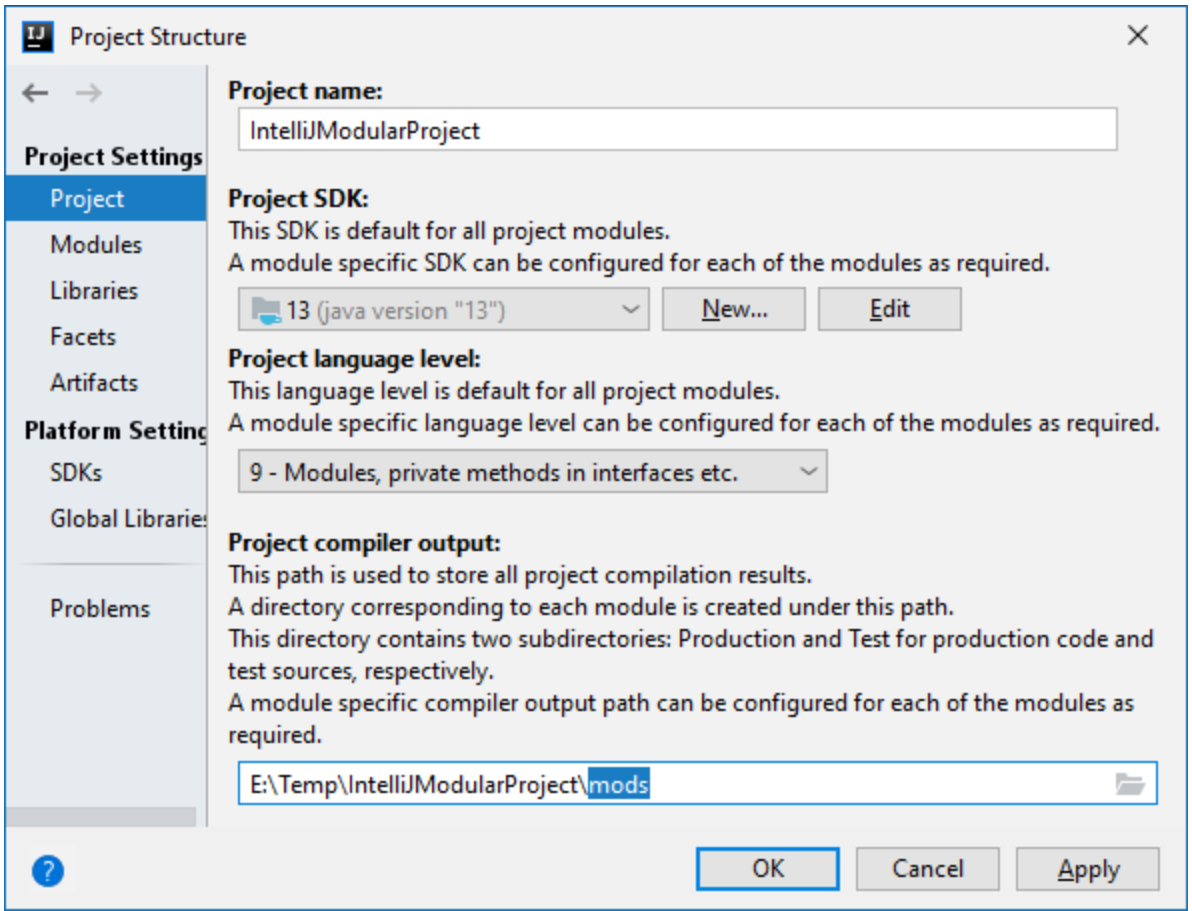
6a.4 Modular Programming with IJ

2.1 Use Project Settings to define

-Project SDK

-Project Language Level (09 Modules, private methods in interfaces etc)

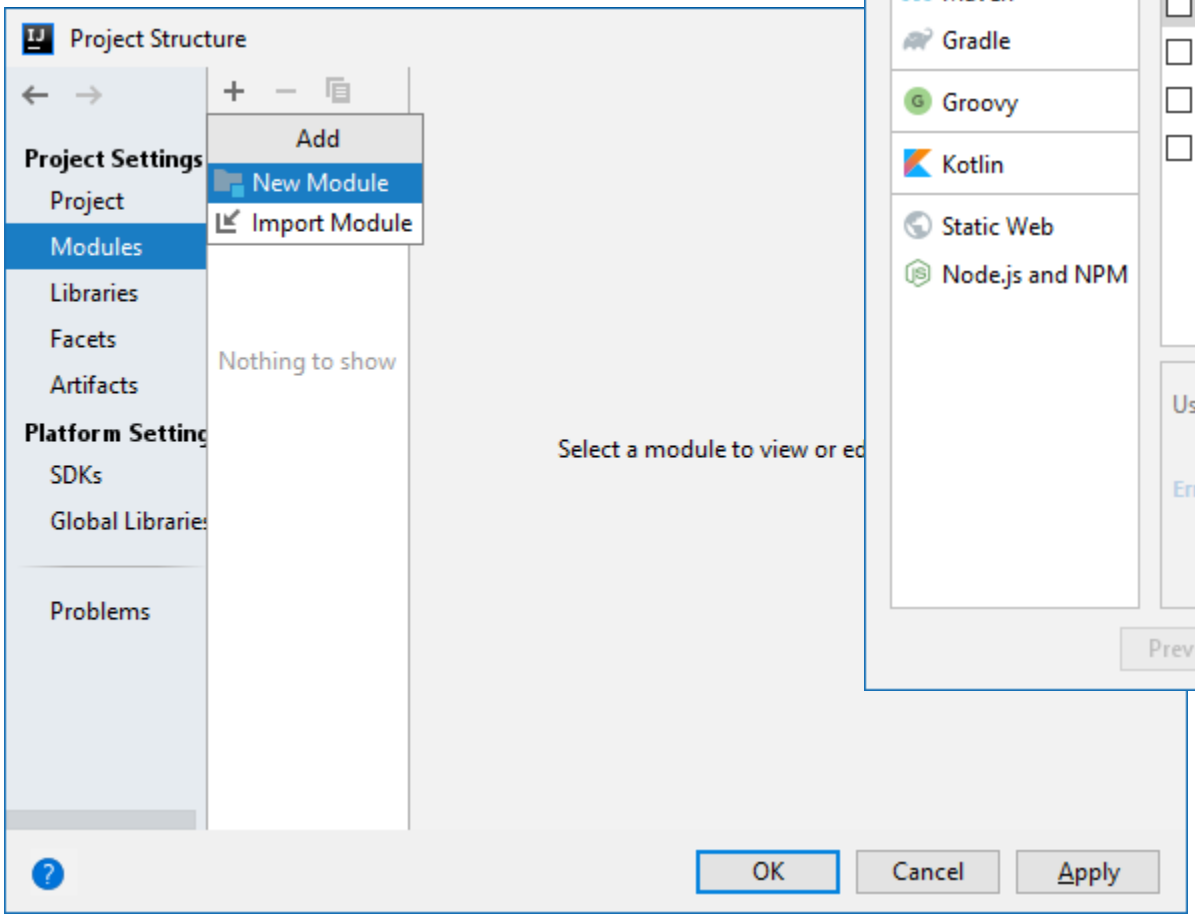
-Project output

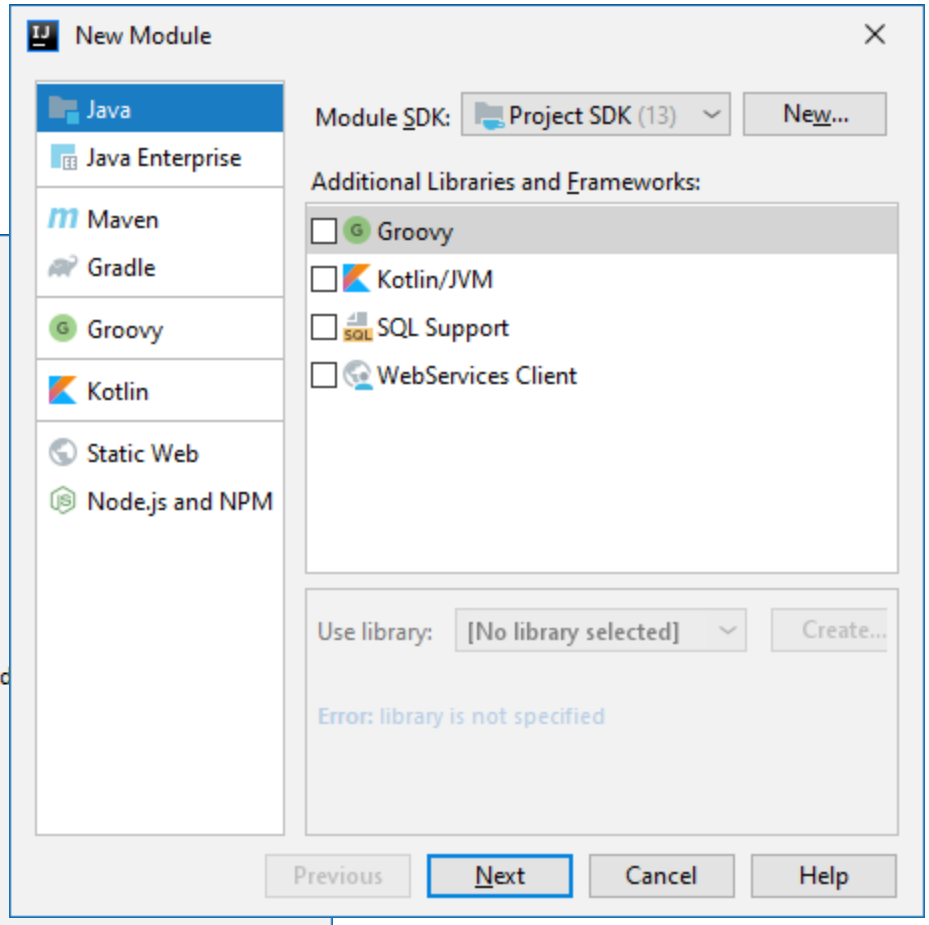


6a.4 Modular Programming with IJ

2.2 Use Project Settings to

–Add new IntelliJ Project module
(it is not a Java Module)

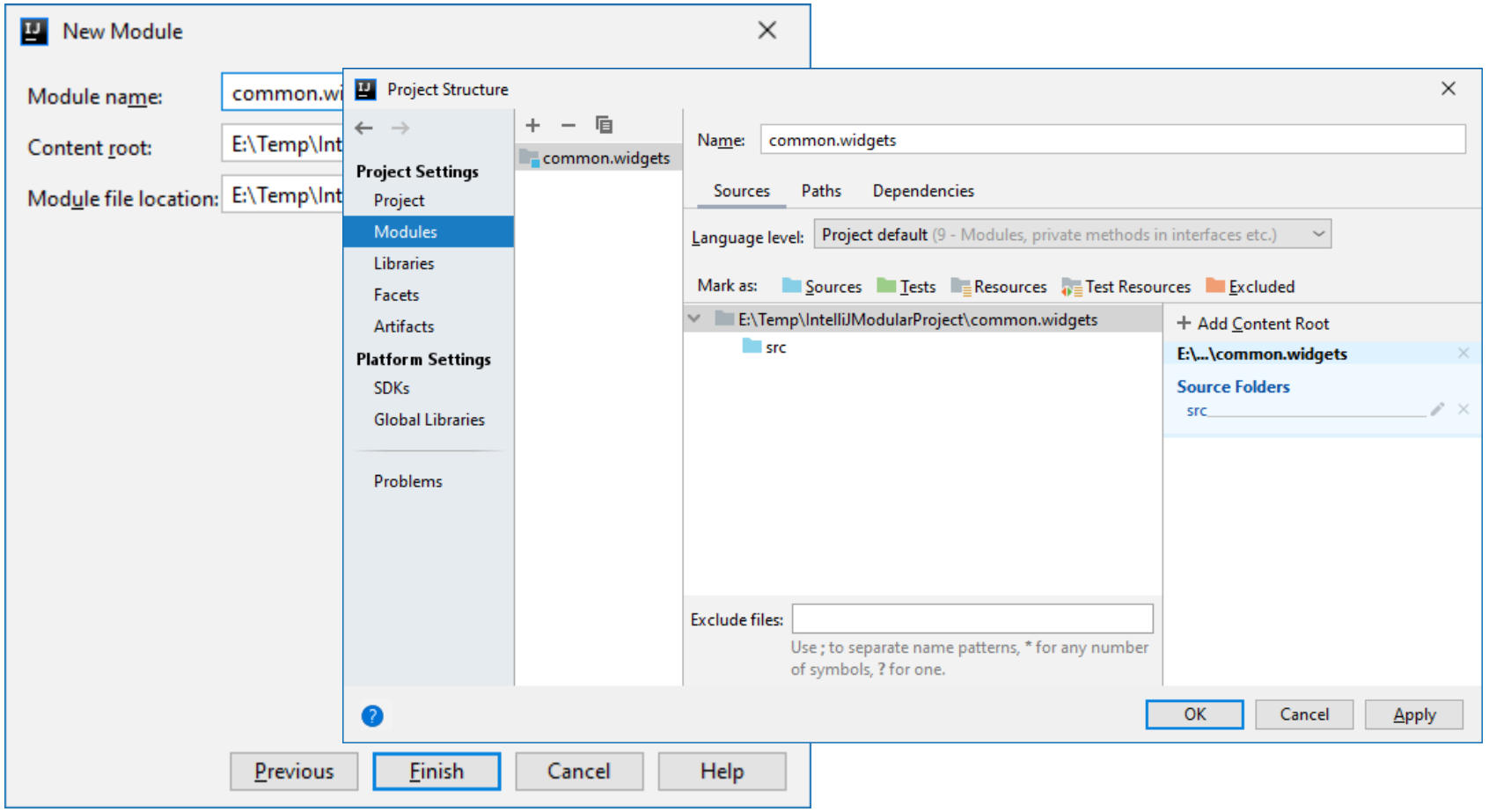




6a.4 Modular Programming with IJ

2.3 Use Project Settings to

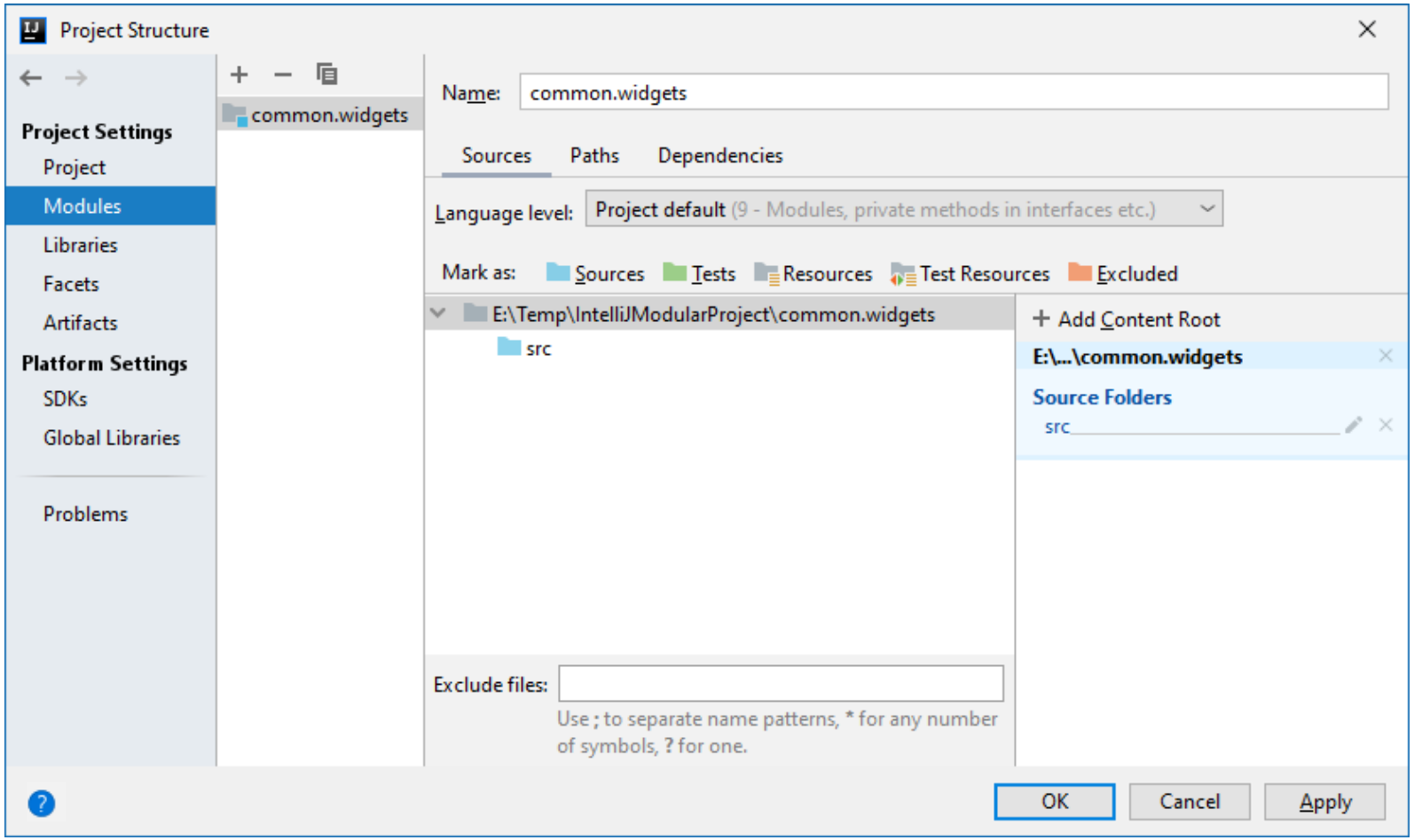
–Enter name for the IntelliJ module



6a.4 Modular Programming with IJ

2.3 Use Project Settings to

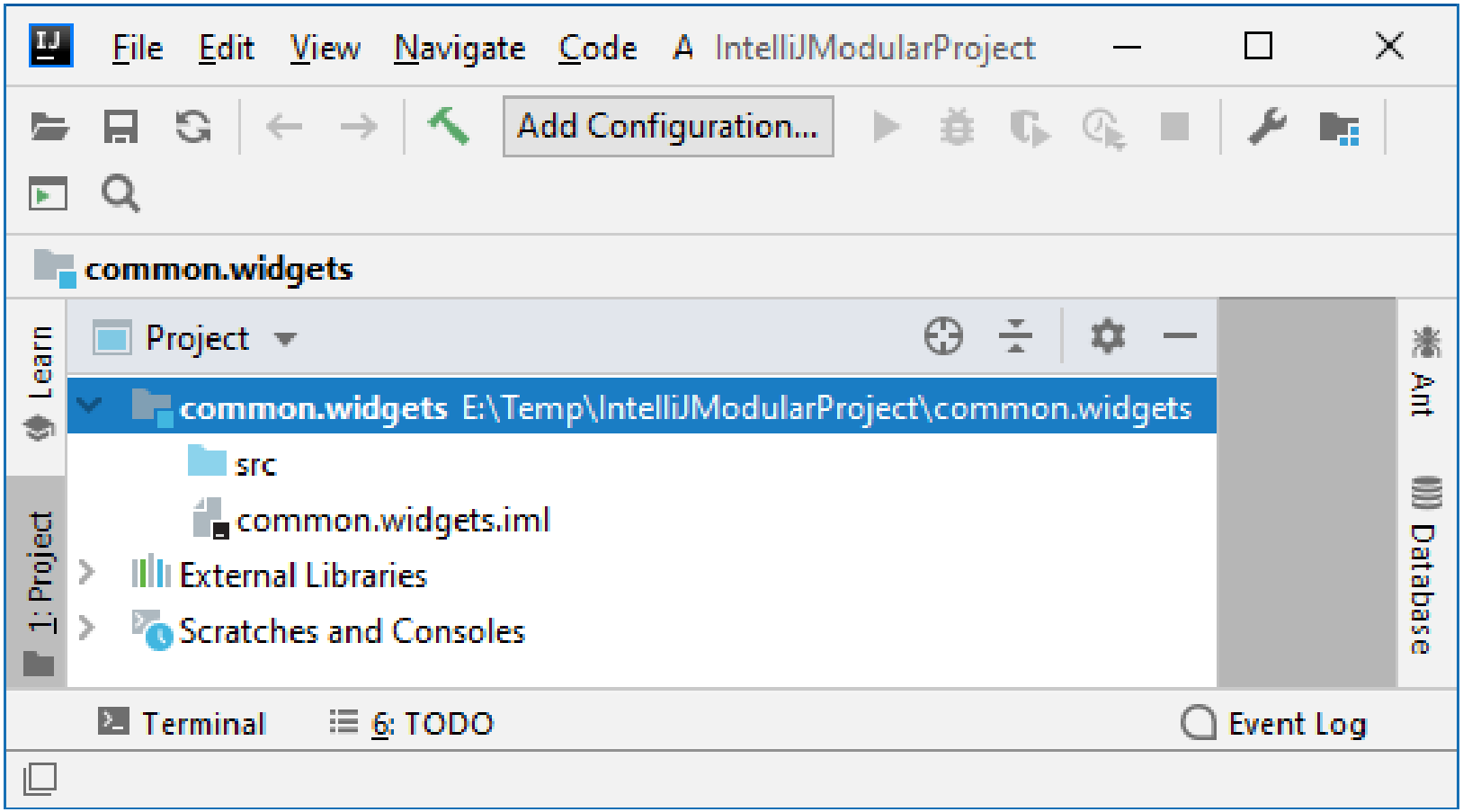
–Enter name for the IntelliJ module



6a.4 Modular Programming with IJ

2.4 Add source code to the IntelliJ module

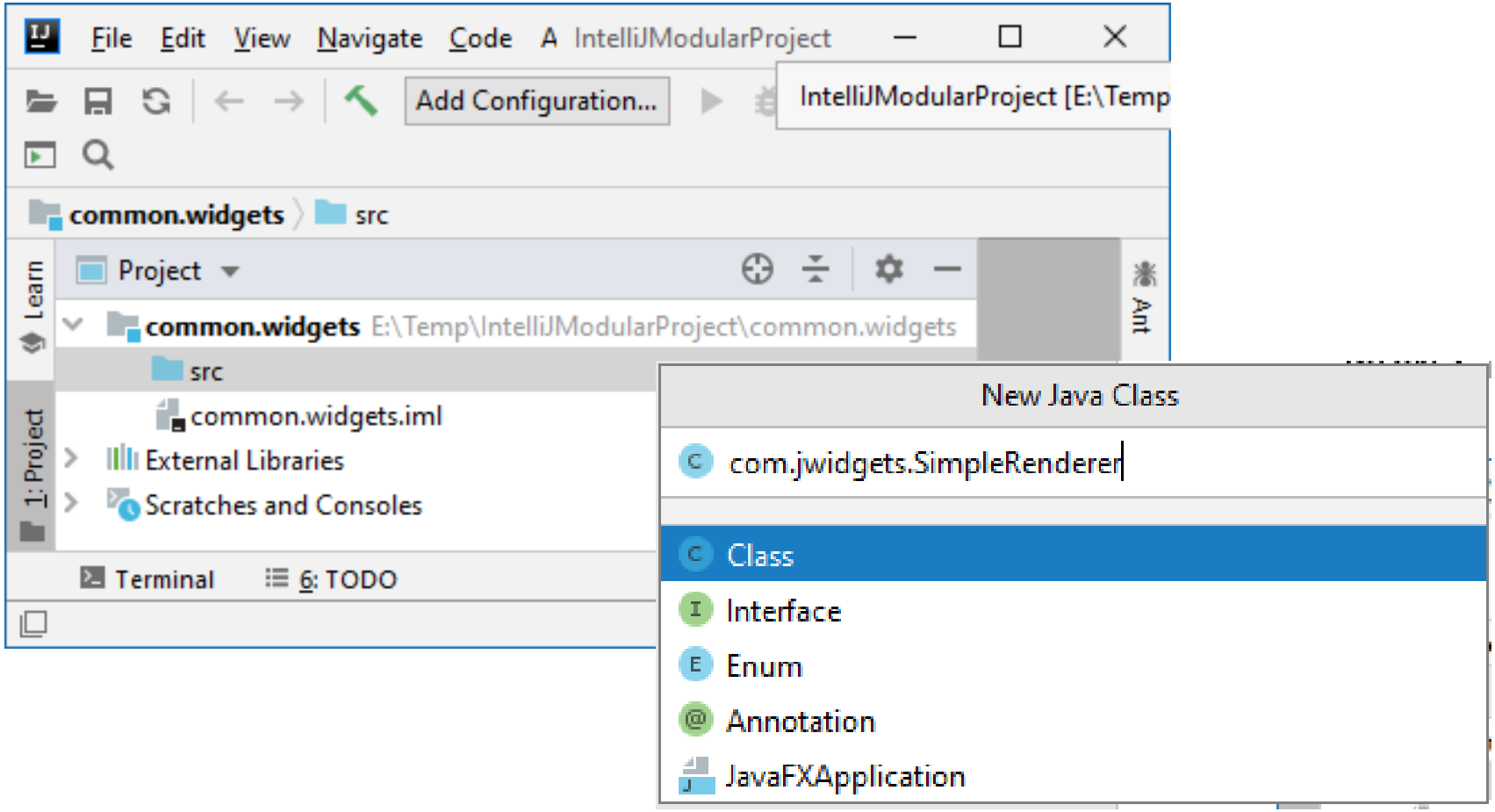
- New Class



6a.4 Modular Programming with IJ

2.4 Add source code to the IntelliJ module

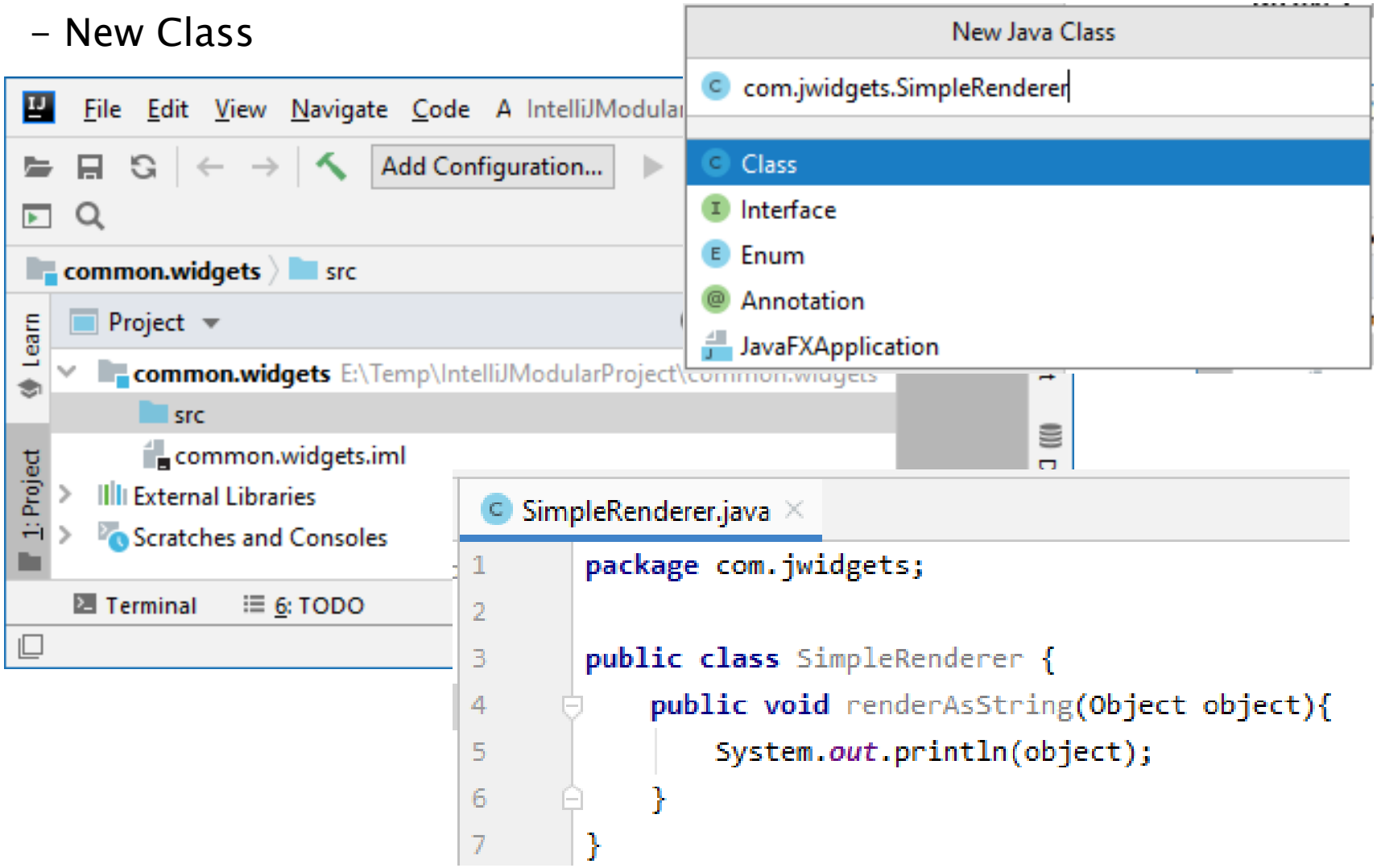
- New Class



6a.4 Modular Programming with IJ

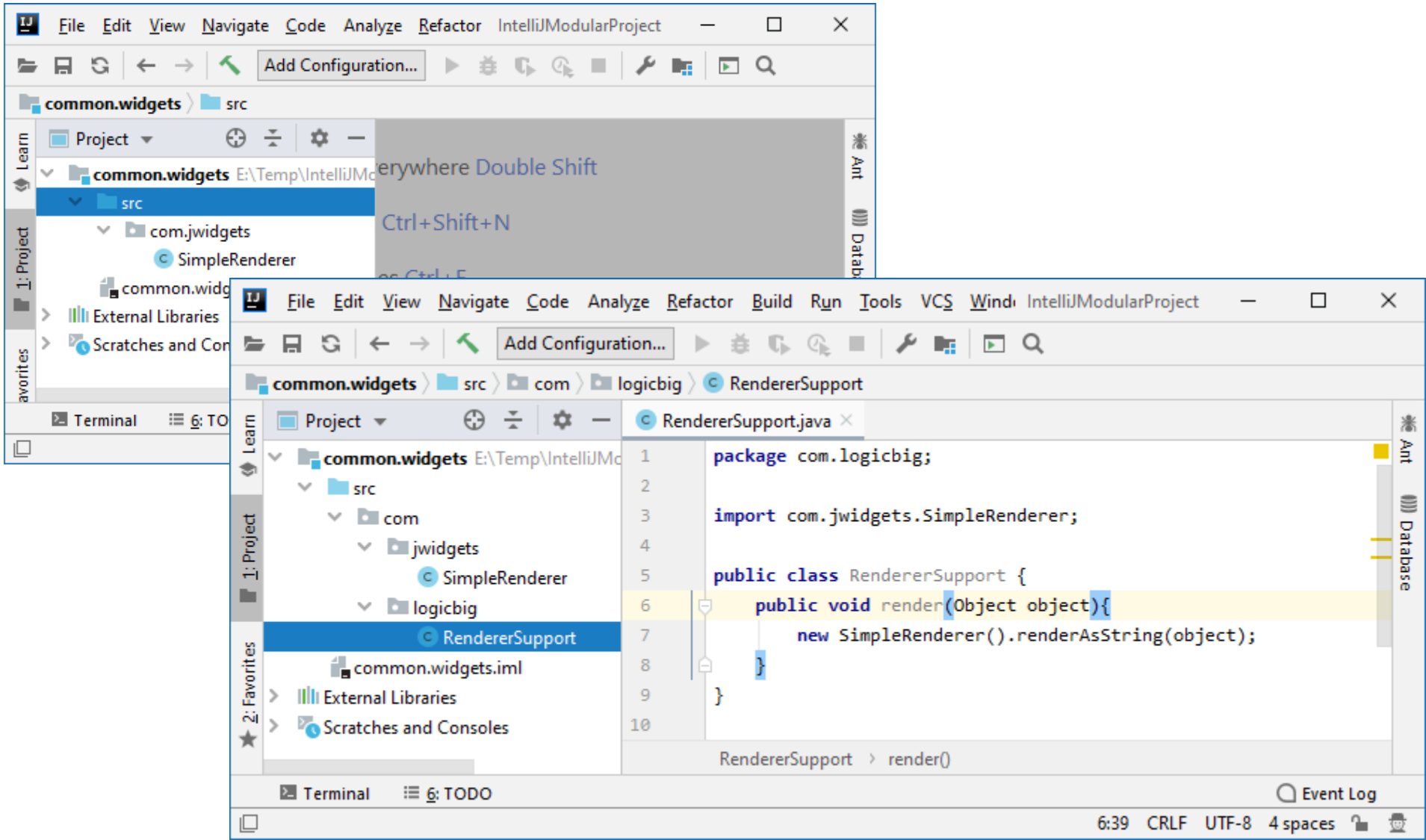
2.4 Add source code to the IntelliJ module

- New Class



6a.4 Modular Programming with IJ

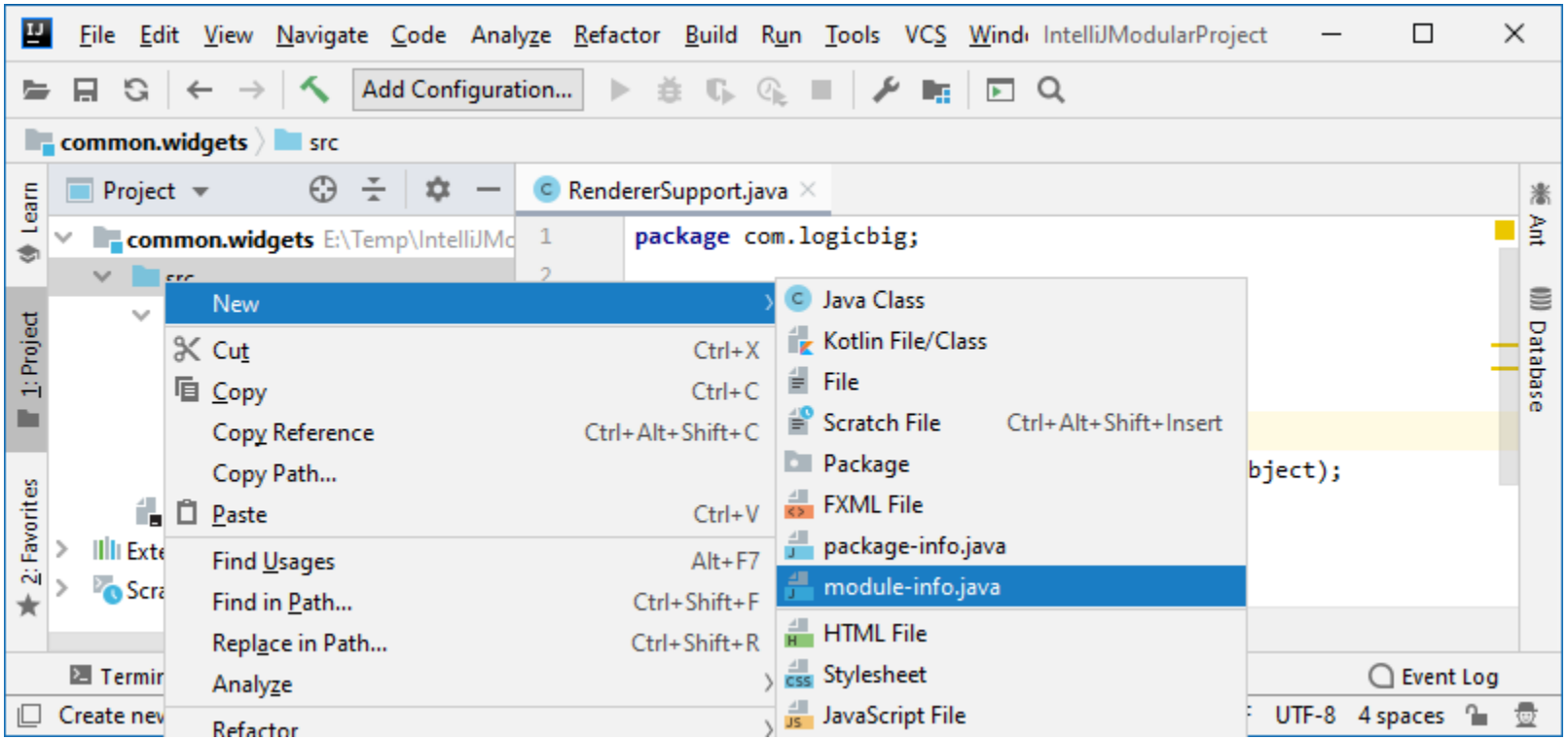
2.4 …. More source code to the IntelliJ module



6a.4 Modular Programming with IJ

2.5 Click on the root folder (`src`) and Add a `module-info.java`

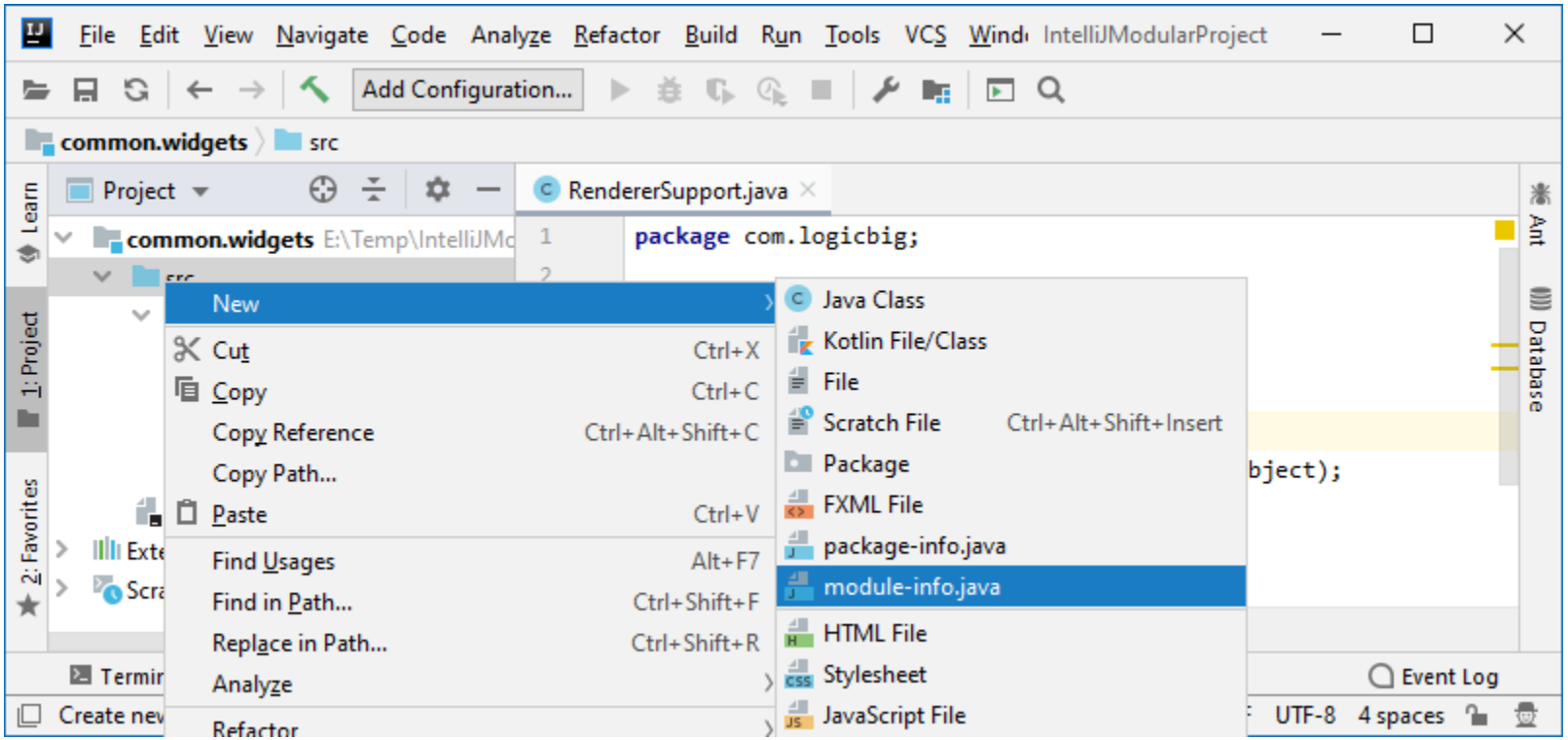
- Add `module-info.java`



6a.4 Modular Programming with IJ

2.5 Click on the root folder (`src`) and Add a `module-info.java`

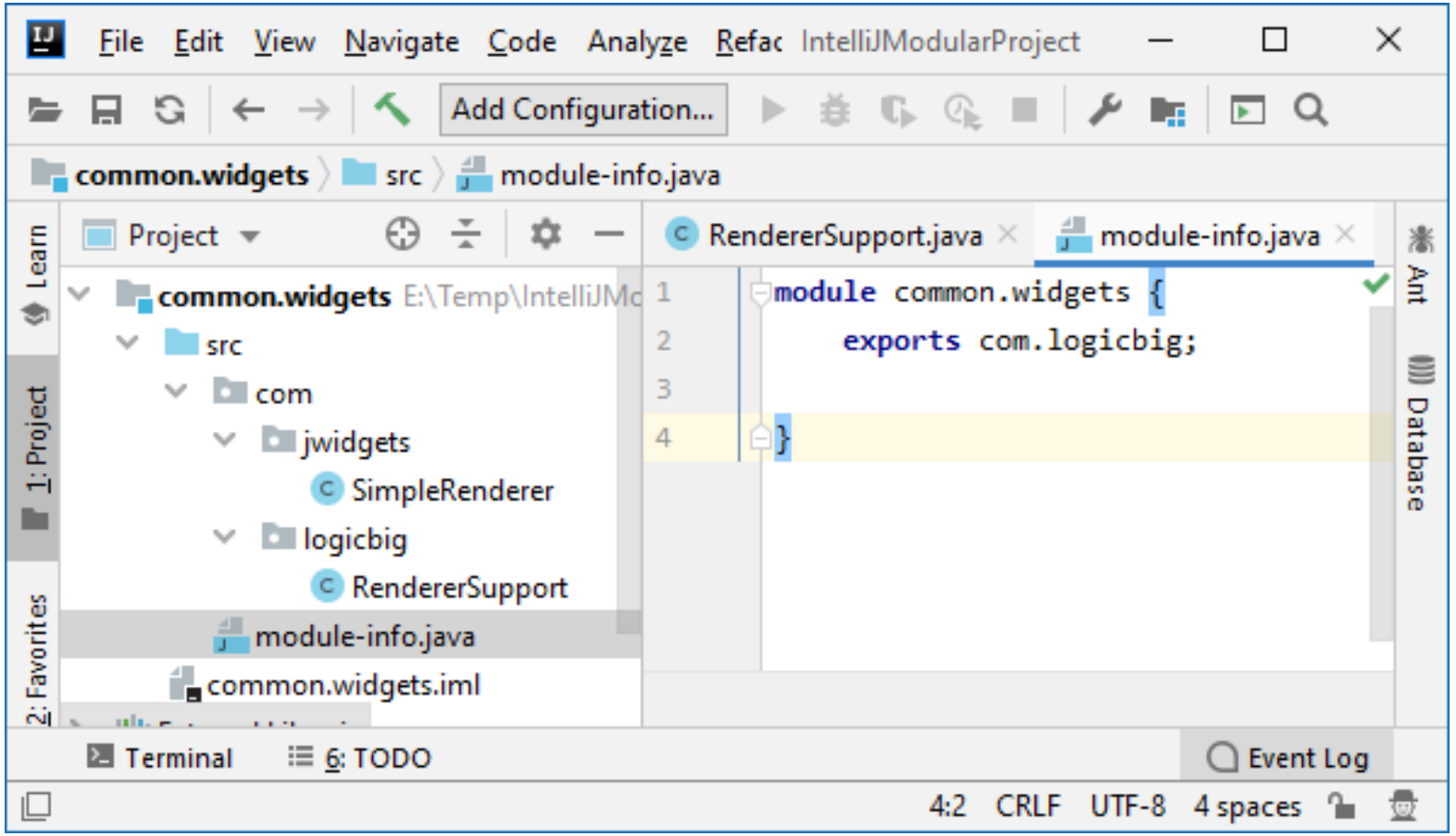
- Add `module-info.java`



6a.4 Modular Programming with IJ

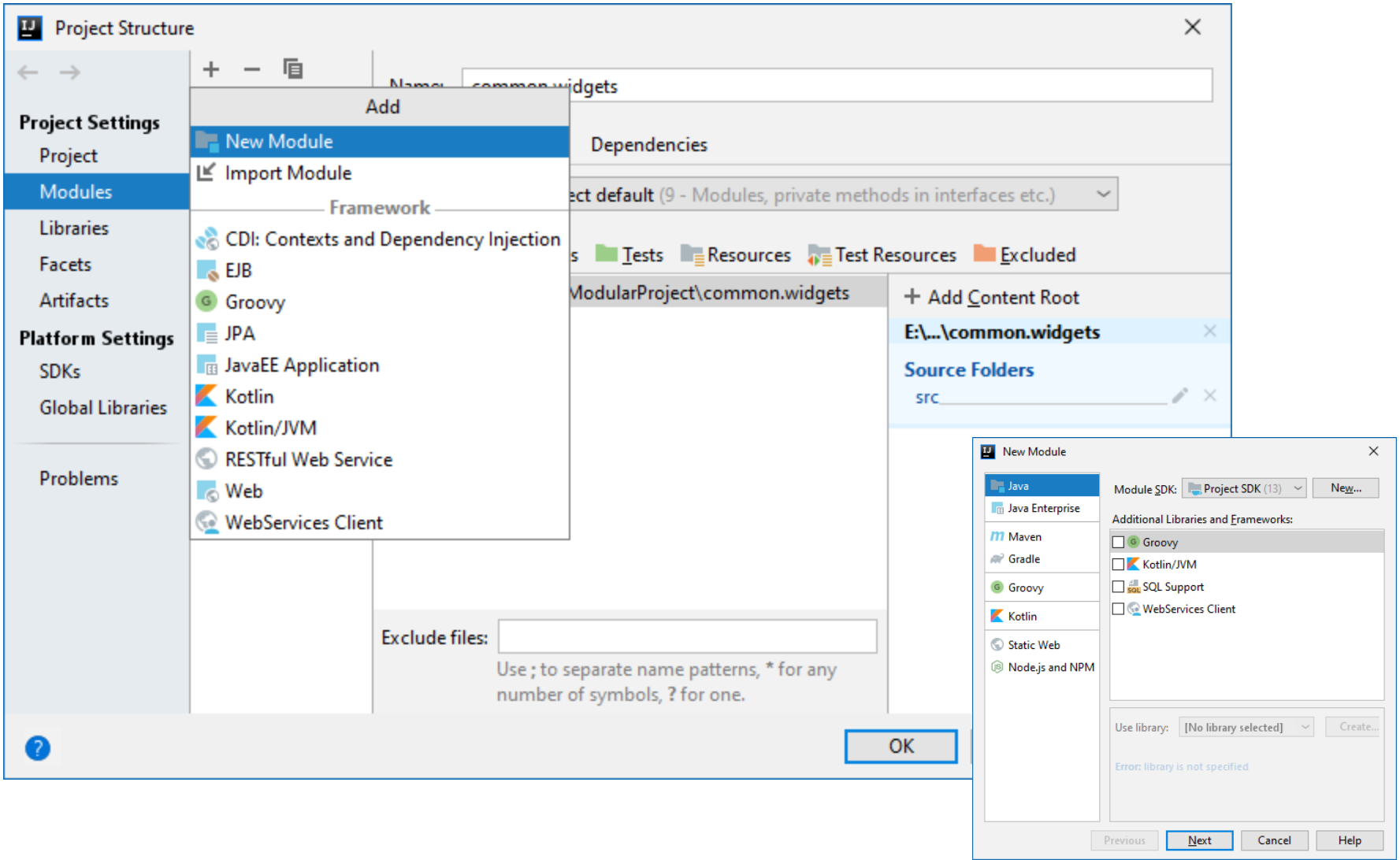
2.5 Click on the root folder (`src`) and Add a `module-info.java`

- Add `module-info.java`



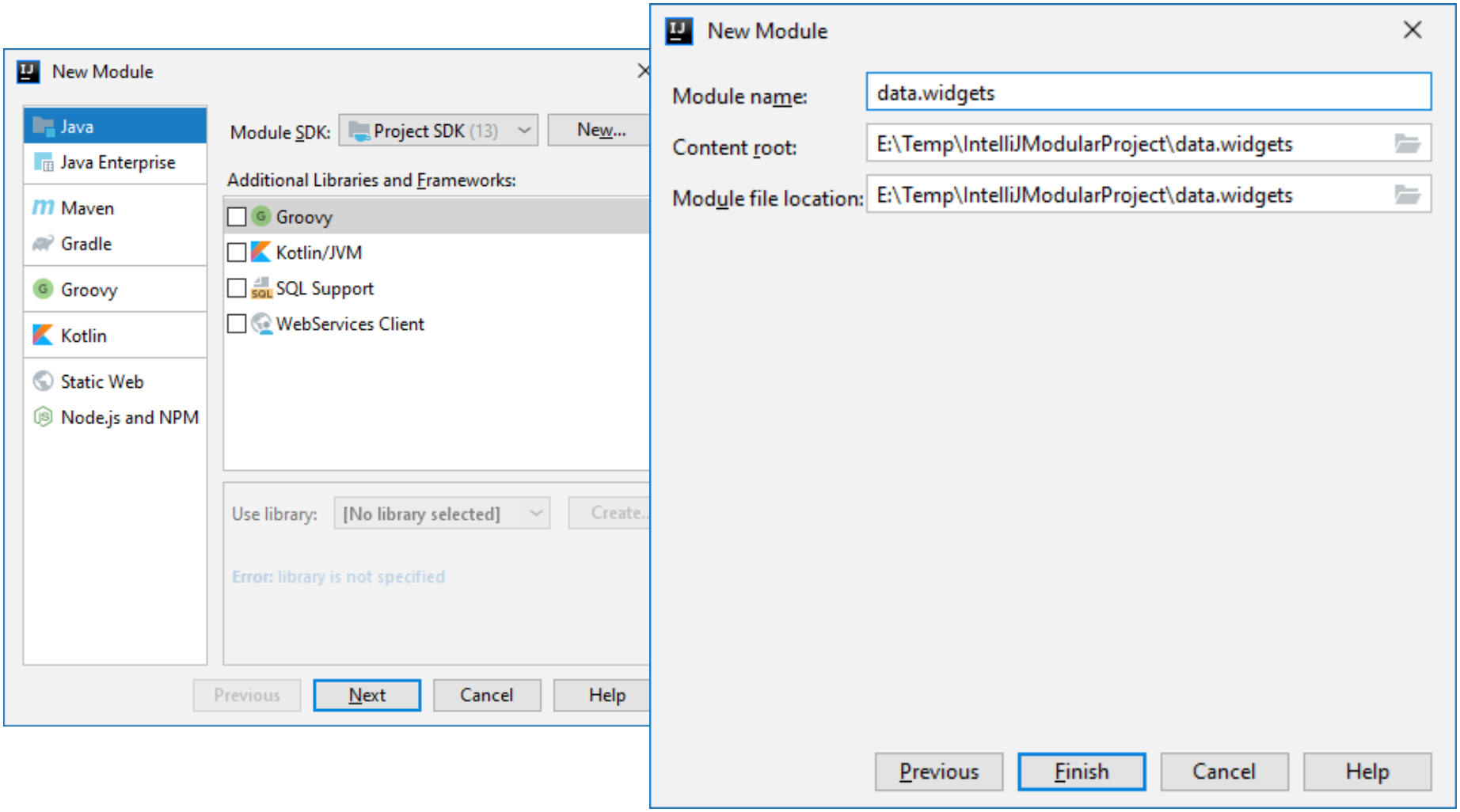
6a.4 Modular Programming with IJ

2.6 Add another IntelliJ module (Shift-Ctrl-Alt-S) **Project Settings**



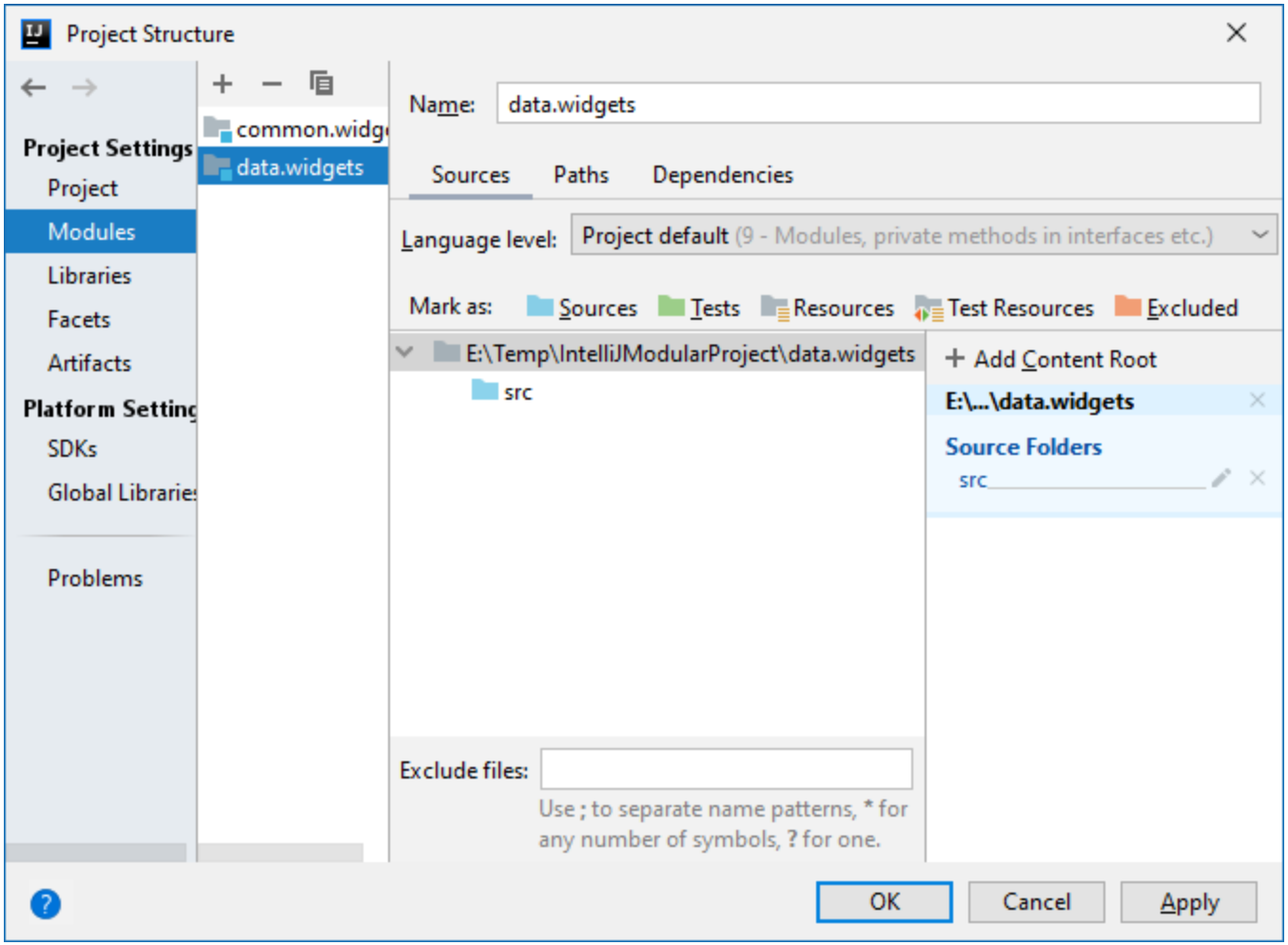
6a.4 Modular Programming with IJ

2.6 … another IntelliJ module (Shift-Ctrl-Alt-S) **Project Settings**



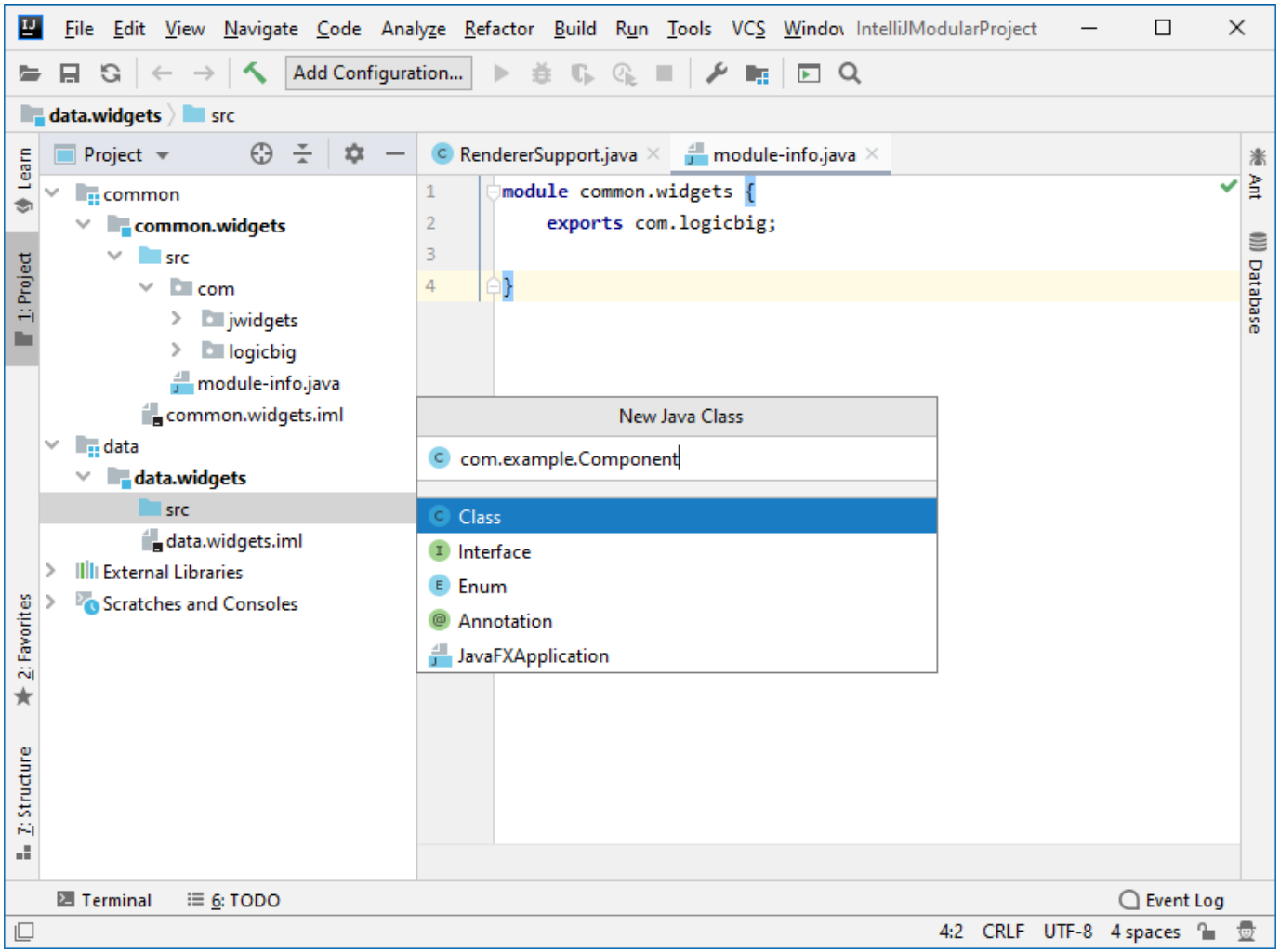
6a.4 Modular Programming with IJ

2.6 … another IntelliJ module (Shift-Ctrl-Alt-S) **Project Settings**



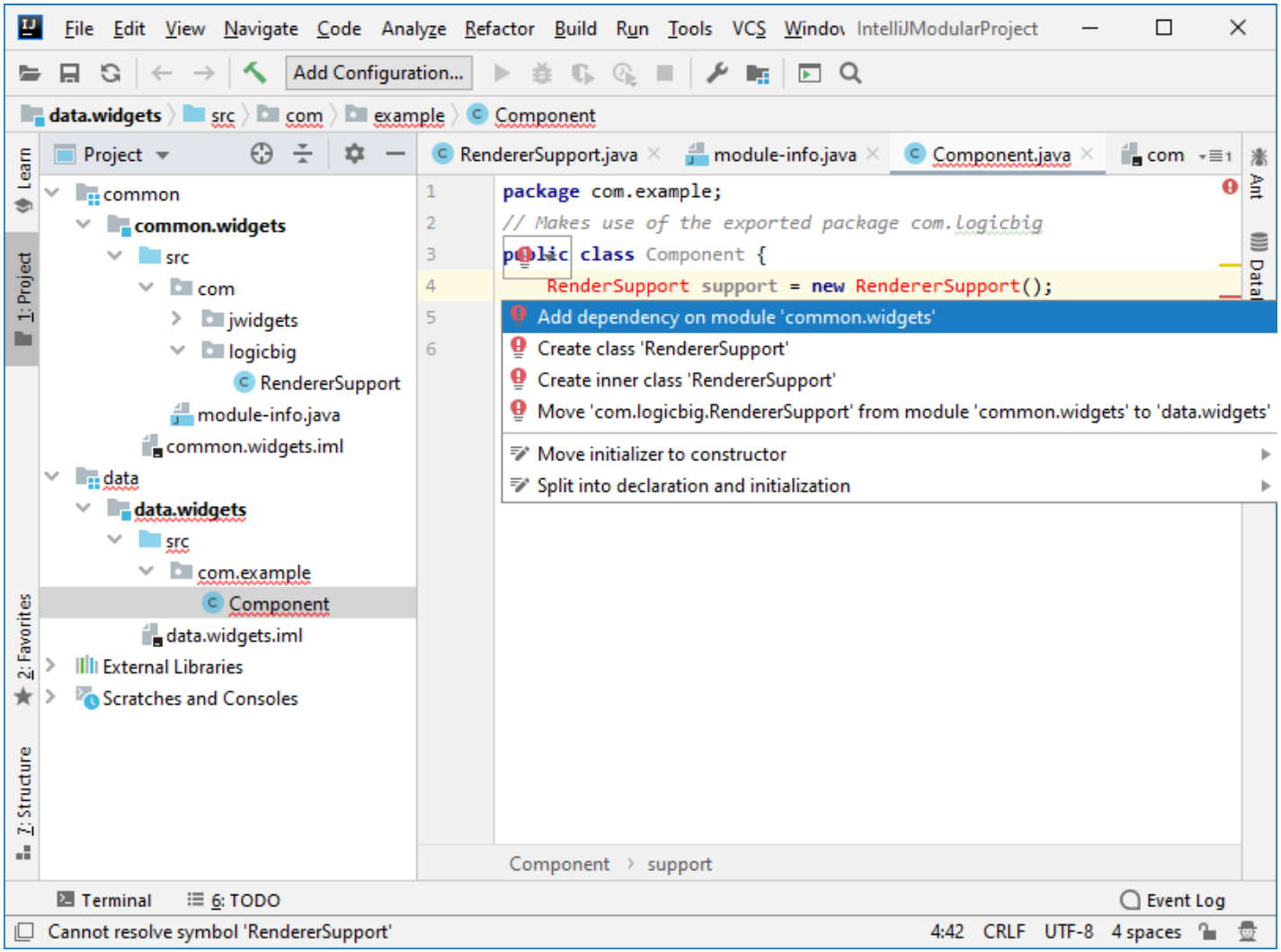
6a.4 Modular Programming with IJ

2.7 Add classes to the new module



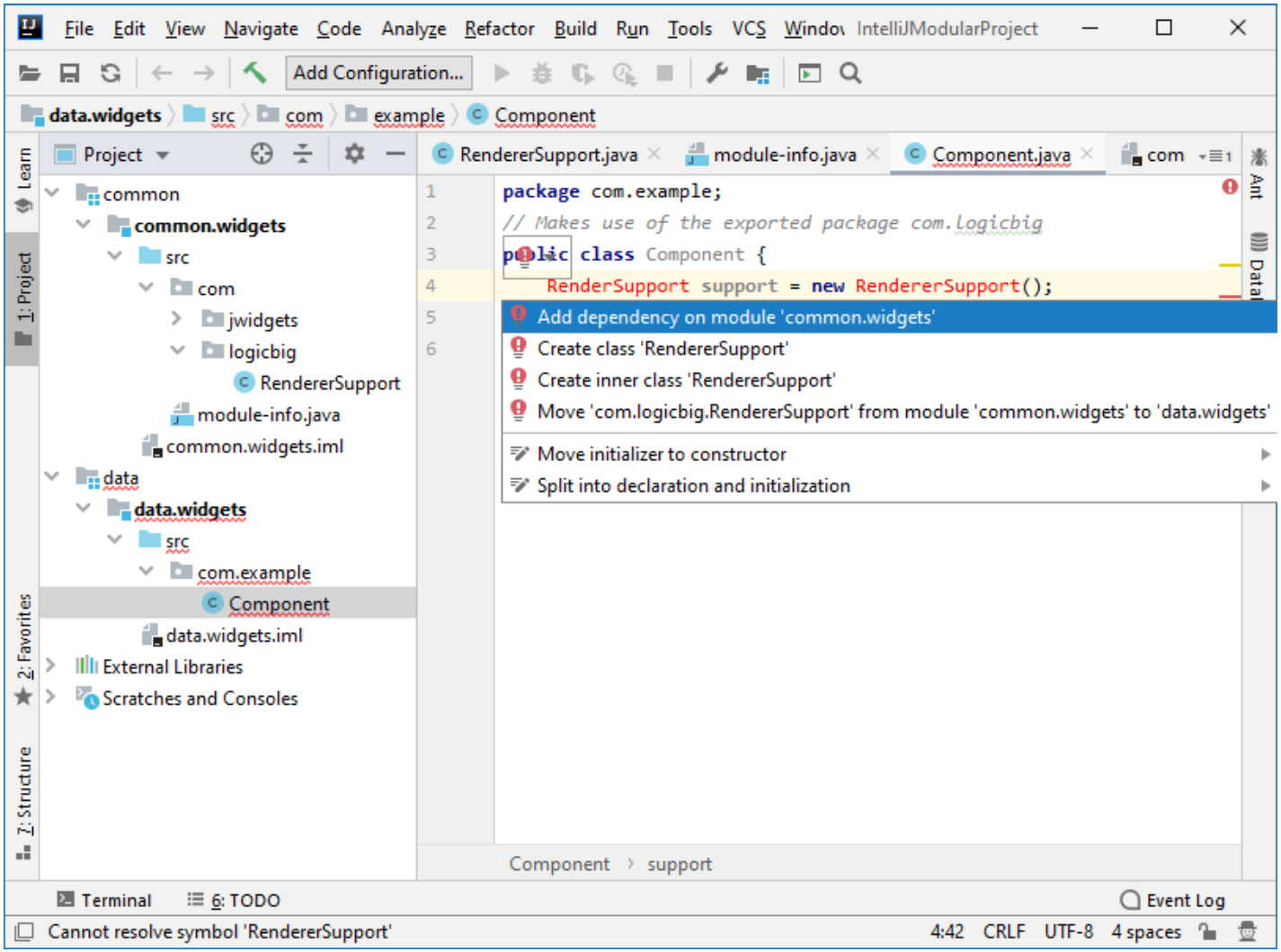
6a.4 Modular Programming with IJ

2.7 …. Add classes to the new module



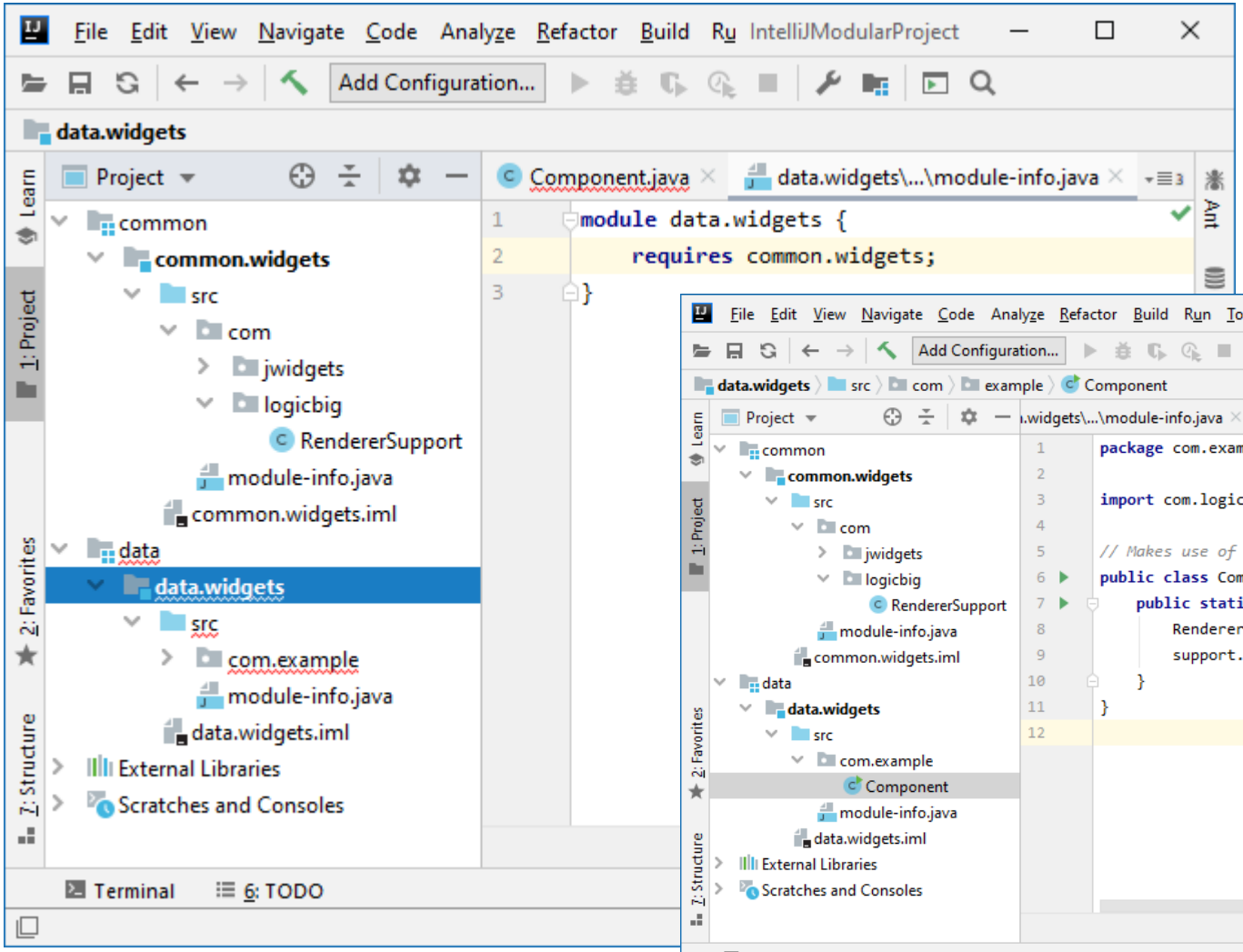
6a.4 Modular Programming with IJ

2.7 …. Add classes to the new module



6a.4 Modular Programming with IJ

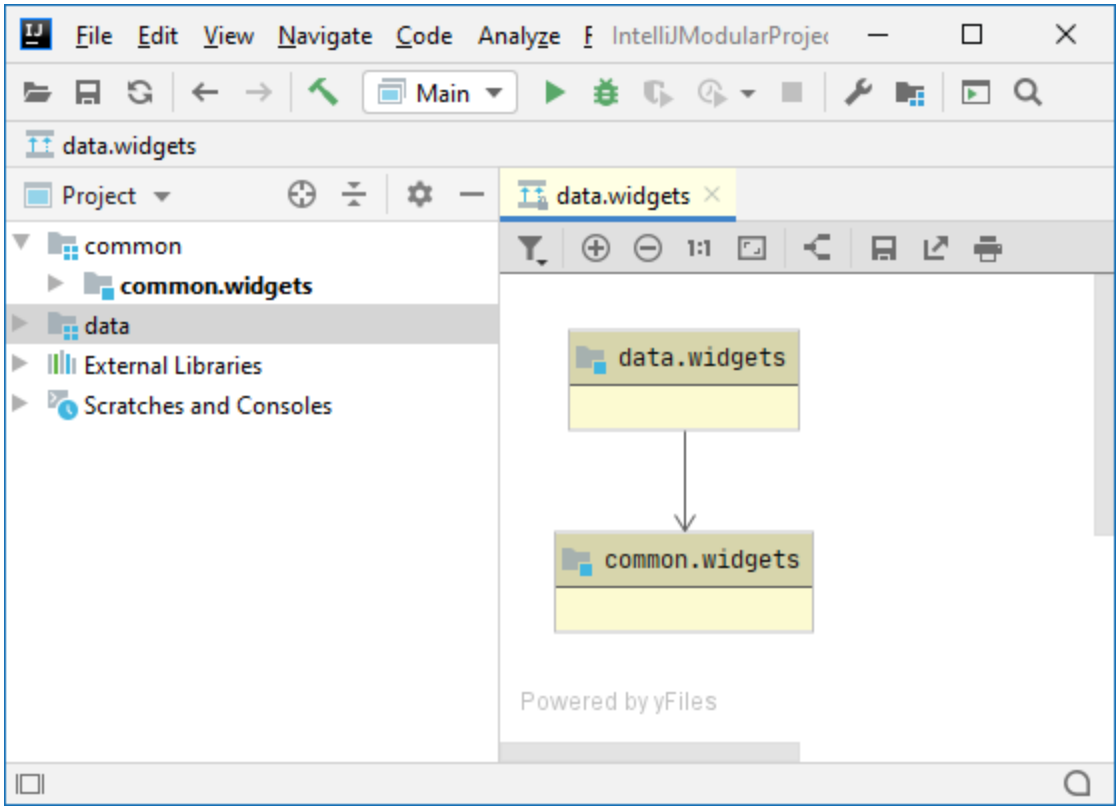
2.8 …. Add module-info.java to the module data.widgets



6a.4 Modular Programming with IJ

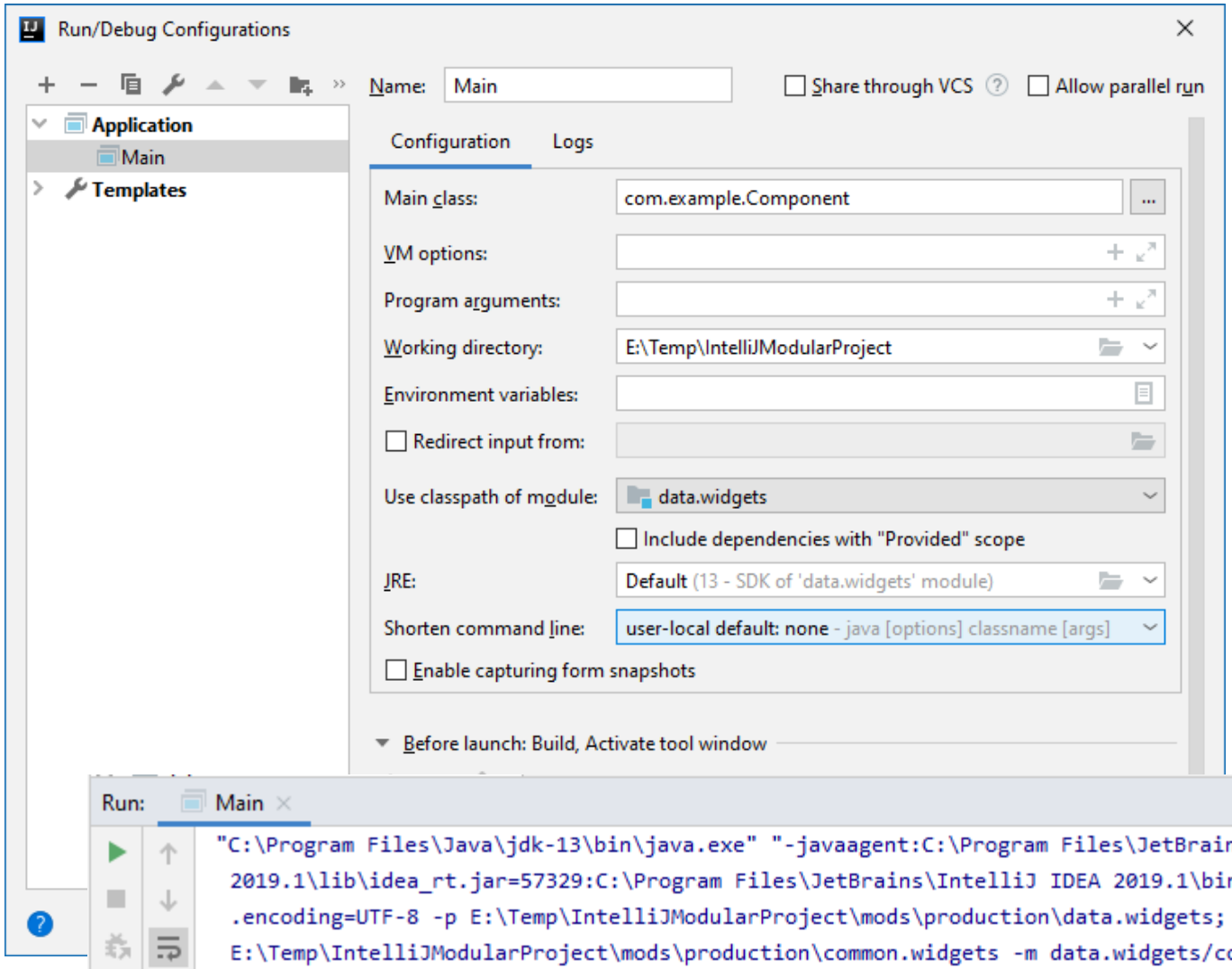
View the **Dependency Diagram** for module **data.widgets**

`Diagram | Show Diagram Ctrl+Shift+Alt+U`



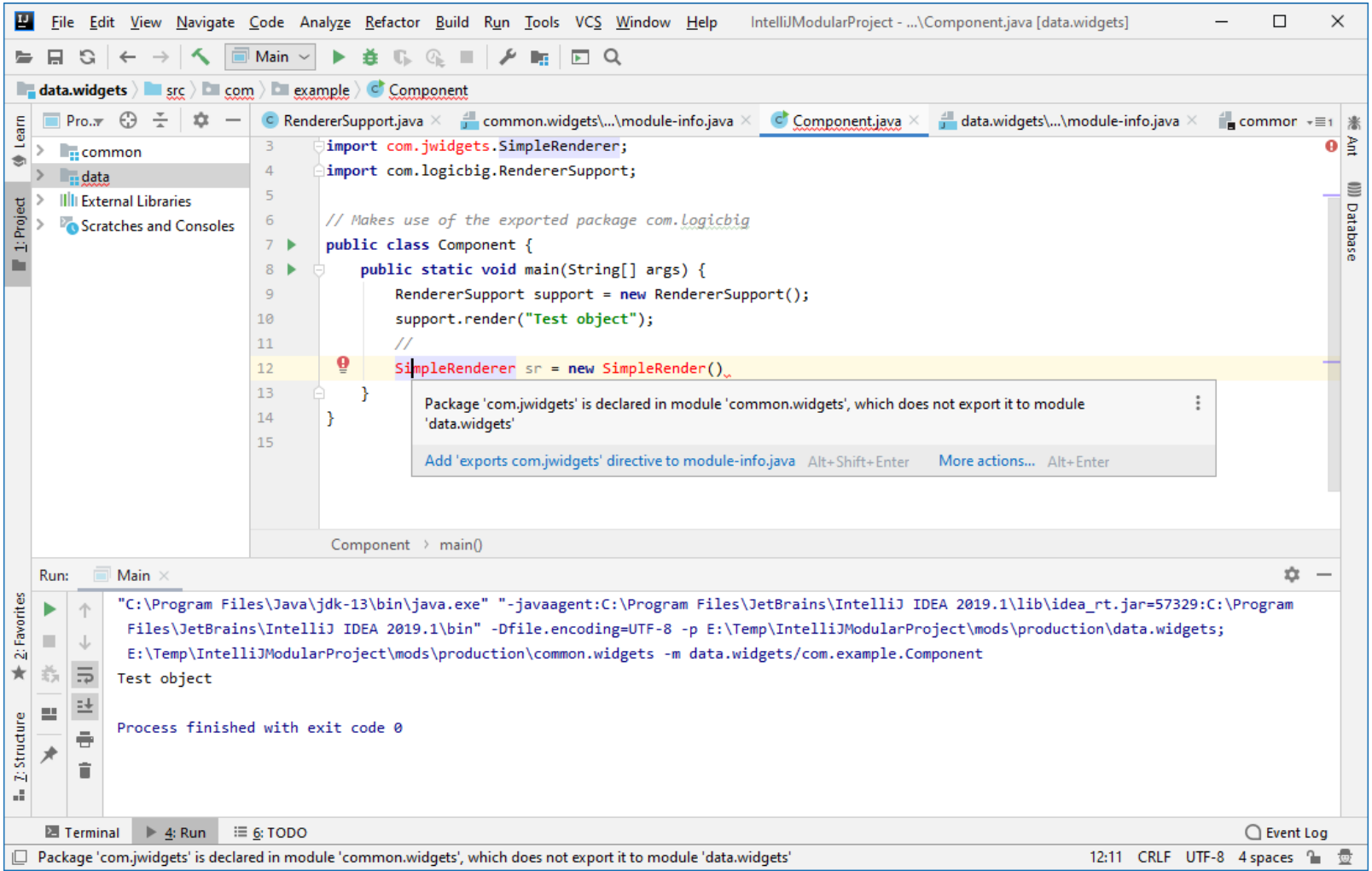
6a.4 Modular Programming with IJ

2.9 Build and Run the project



6a.4 Modular Programming with IJ

Experiment to use a **`class SimpleRender`** from a package not exported in **`module-info.java`**. It is not allowed to use classes from packages that are not exported.



6a.4a The Unnamed Module

In Java 9 and later editions, **all code is required to be placed in modules**. When you execute code that is not in a module, the code is loaded from the **classpath** and placed in the **unnamed** module.

A class which is not a member of a **'named module'** is considered to be a member of a special module known as the **unnamed module**.

The unnamed module concept is **like the unnamed package** (the default package). The unnamed module is **not a real module**. It can be considered as the **default module** which does not have a name.

All classed compiled in Java 8 and older versions, which are not yet migrated to modules, also belong to the unnamed module when run in Java 9 or later editions.

6a.4a The Unnamed Module

In Java 9 and later editions, **all code is required to be placed in modules**. When you execute code that is not in a module (missing module descriptor), the code is loaded from the **classpath** and placed in the **unnamed** module.

A class which is not a member of a **'named module'** is considered to be a member of a special module known as the **unnamed module**.

The unnamed module concept is **like the unnamed package** (the default package). The unnamed module is **not a real module**. It can be considered as the **default module** which does not have a name.

All classed compiled in Java 8 and older versions, which are not yet migrated to modules, also belong to the unnamed module when run in Java 9 or later editions.

6a.4a The Unnamed Module

In case we place an owning module onto the **classpath** the module system reacts as follows.

Since everything needs to be a module **the module system simply creates one, the unnamed module, and puts everything in it that it finds on the class path**. Inside the unnamed module everything is much like it is today and JAR continues to exist. Because the unnamed module is synthetic, the JPMS has no idea what it might export so **it simply exports everything** – at **compile** and at **run** time.

6a.4a The Unnamed Module

If any JAR on the classpath should accidentally contain a module descriptor, this mechanism will simply ignore it.

Hence, the owning module gets demoted to a regular JAR and its code ends up in a module that exports everything:

public: ✓ protected: ✓ default: ✓ private: ✓

Without touching the owning module, so we can do this to modules we have no control over. Small caveat- **we can not require the unnamed module** so there is no good way to compile against the code in the owning module from other modules.

6a.4a The Unnamed Module

- **The unnamed module 'requires' every other named modules automatically.**

That means **all classes within the unnamed module can read all other module (named or unnamed) without any explicit declaration of any kind.**

That also means that **older classes (written prior to Java 9) can read all modules defined by the new Module system.** Hence, all such classes will continue to compile and run in Java SE 9 without any modification.

6a.4 The Unnamed Module

- **The unnamed module 'exports' all its packages.**

That means different **JAR applications which do not contain module or which are compiled in the older versions**, will continue to use each other's dependencies as it is.

The **packages exported by unnamed module can only be read by another unnamed module**.

Note: **It is not allowed a named module to read (requires) the unnamed module**. Of course, to explicitly use 'requires' clause in a **`module-info.java`** or use a command line option to add the module, we need a module name.

6a.4a The Unnamed Module

Let's create a project without a module.

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help ProjectWithUnnamedModule

Main

default › src › com › test › Main

Project

default E:\Temp\ProjectWithUnnamedModule\default
src
com.test
Main
default.iml
External Libraries
Scratches and Consoles

Main.java

```
1  package com.test;
2
3  public class Main {
4      public static void main(String[] args) {
5          Module module = Main.class.getModule();
6          System.out.println("Module: "+module);
7          System.out.println("Name: " + module.getName());
8          System.out.println("isNamed: " + module.isNamed());
9          System.out.println("Descriptor: " + module.getDescriptor());
10     }
11 }
12
```

Main

Run: Main

```
E:\Temp\ProjectWithUnnamedModule\out\production\default com.test.Main
Module: unnamed module @723279cf
Name: null
isNamed: false
Descriptor: null

Process finished with exit code 0
```

Terminal 0: Messages 4: Run 6: TODO Event Log

Build completed successfully in 5 s 173 ms (2 minutes ago) 3:14 CRLF UTF-8 4 spaces

6a.4a The Unnamed Module

In above example, we used the method java.lang.Class.Main.class.getModule()

which returns an instance of java.lang.Module (also introduced in Java 9).

The **returned instance module** of the Module **represents a module that this class is a member of.**

The method module.getDescriptor() returns an **ModuleDescriptor** object which typically represents details of **module-info.class**.

Other methods we used above are self-explanatory.

6a.4b Automatic Modules

There are enormous numbers of preexisting libraries that you can use in your apps. Many of these **are not yet modularized**, but to facilitate migration, **you can add any library's JAR file to an app's `module path` then use the packages in that JAR**.

When you do, the JAR file implicitly becomes an **automatic module** and can be specified in the module declaration's requires directive.

The **module name is derived** from the **Automatic-Module-Name** MANIFEST.MF entry or **the filename of the JAR**. They **can read all modules**, **including the unnamed module**, and are a good way to start the migration to modularity.

6a.4b Automatic Modules

One scenario in using Modules is to strip the owning module of its descriptor and still put it on the module path. For each regular JAR on the module path the JPMS creates a new module, names it automatically based on the file name, and exports all its contents. Since all is exported, we get the same result as with the unnamed module:

public: ✓ protected: ✓ default: ✓ private: ✓

Nice. **The central advantage of automatic modules over the unnamed module is that modules can require it, so the rest of the application can still depend on and compile against it**, while the intruder can use reflection to access its internals.

6a.4b Automatic Modules

One downside is that **the automatic module's internals become available at run time to every other module in the system**. Unfortunately, the same is true at compile time unless we manage to compile against the proper owning module and then rip out its descriptor on the way to the launch pad. This is tricky, and error-prone.

6a.4b Automatic Modules

An **automatic module**:

- **Automatic modules** are JAR files on the **module-path without the** `module-info.class` file
- **Implicitly exports** all the package's types so any module that reads the automatic module (including the **unnamed module**) has access to the public types in the automatic module's packages
- **Implicitly reads** (requires) all other modules, including other automatic modules and the unnamed module, so **an automatic module has access to all the public types exposed by the system's other modules.**

6a.4c Automatic Modules

Module types Summary:

- **Java SE and JDK modules** are the modules provided by the JDK: java.base, java.xml, etc. They export all the packages you could access previously, like java.util.
- **Named application modules** are your application modules that contain the module-info.class file; they have to explicitly state which modules they depend on, including what packages and services they need. They cannot read the unnamed module. (see below)
- **Automatic modules** are jar files on the module-path without the module-info.class file. The module name is derived from the Automatic-Module-Name MANIFEST.MF entry or the filename of the jar; they can read all modules, including the unnamed module, and are a good way to start the migration to modularity.
- **Unnamed module** contains all the jars and classes on the classpath; can read all modules.

6a.5 Modular Programming with IJ

Note: **Import an existing module to the project. (the imported module is NOT copied in this project)**

You can **import** a module and **add** it to your **IntelliJ project**, or you can **use the project as a shared view for multiple individual modules**. This is useful if you have several projects and your workflow requires that you switch between them.

1. To add an existing module to your project, import the **.iml** file:
2. From the main menu, select **File | New | Module from Existing Sources.**
3. In the dialog that opens, specify the path the **.iml** file of the module that you want to import, and click **Open.**
4. **On this stage, the module is not a part of the project.**
5. In the **Project** tool window, **drag the imported module** to the top-level directory to add it to the project.

6a.5 Modular Programming with IJ

Module Path

As seen in above example, Java Module system uses module path instead of class path to locate the modules. The module path solves many issues of class path; most notably issues like making no distinction between the loaded artifacts, no reporting of missing artifacts in advance, allowing conflicts between same classes of different version residing in the different jars on the class path. As opposed to class path, the module path locates whole modules rather than individual classes.

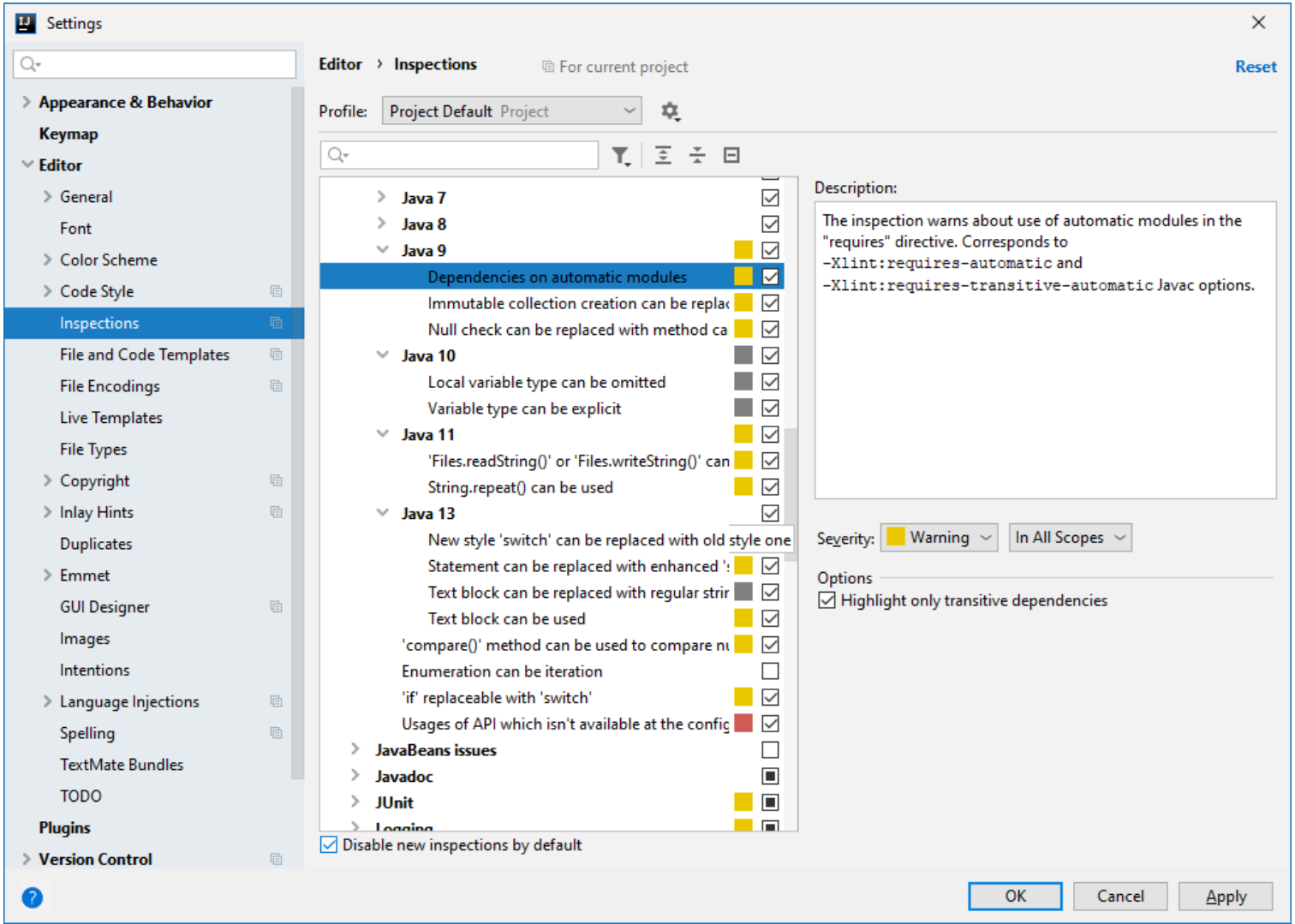
Denote the module path for projects with unnamed modules in VM options using

--module-path <path to module> or

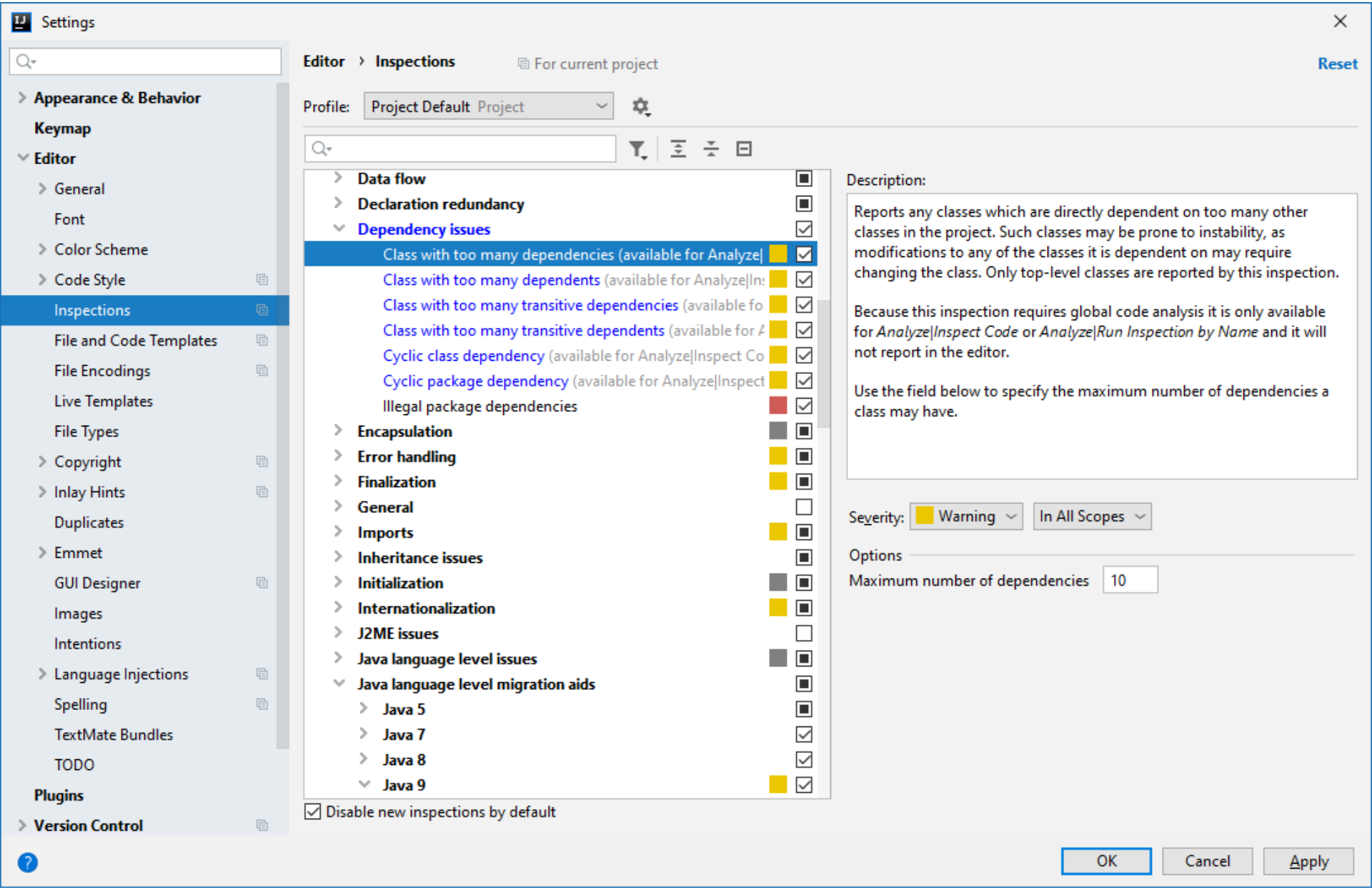
-p <path to module>

6a.6 Hints with IJ

IntelliJ IDEA has inspections (**Analyze->Inspect code**) to help migrate code to Java 9. For example, you can highlight the use of automatic modules.



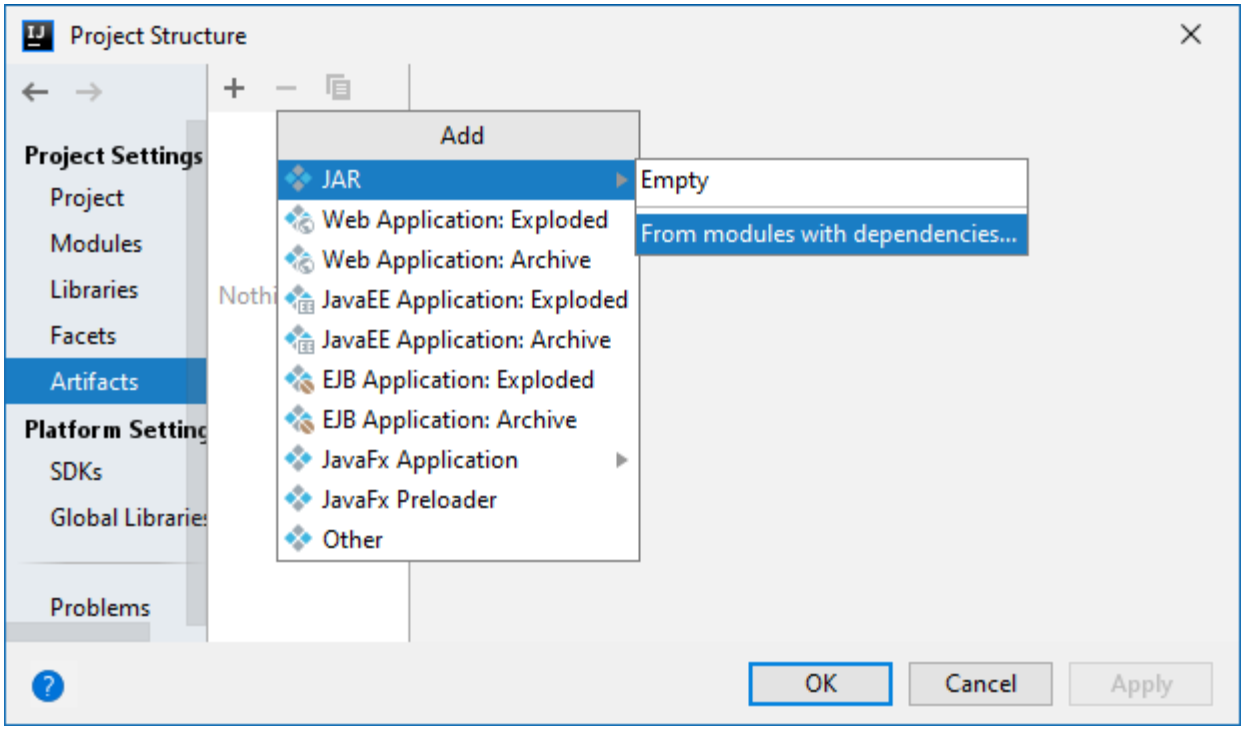
6a.6 Hints with IJ



6a.7 Add modules to a modular JAR

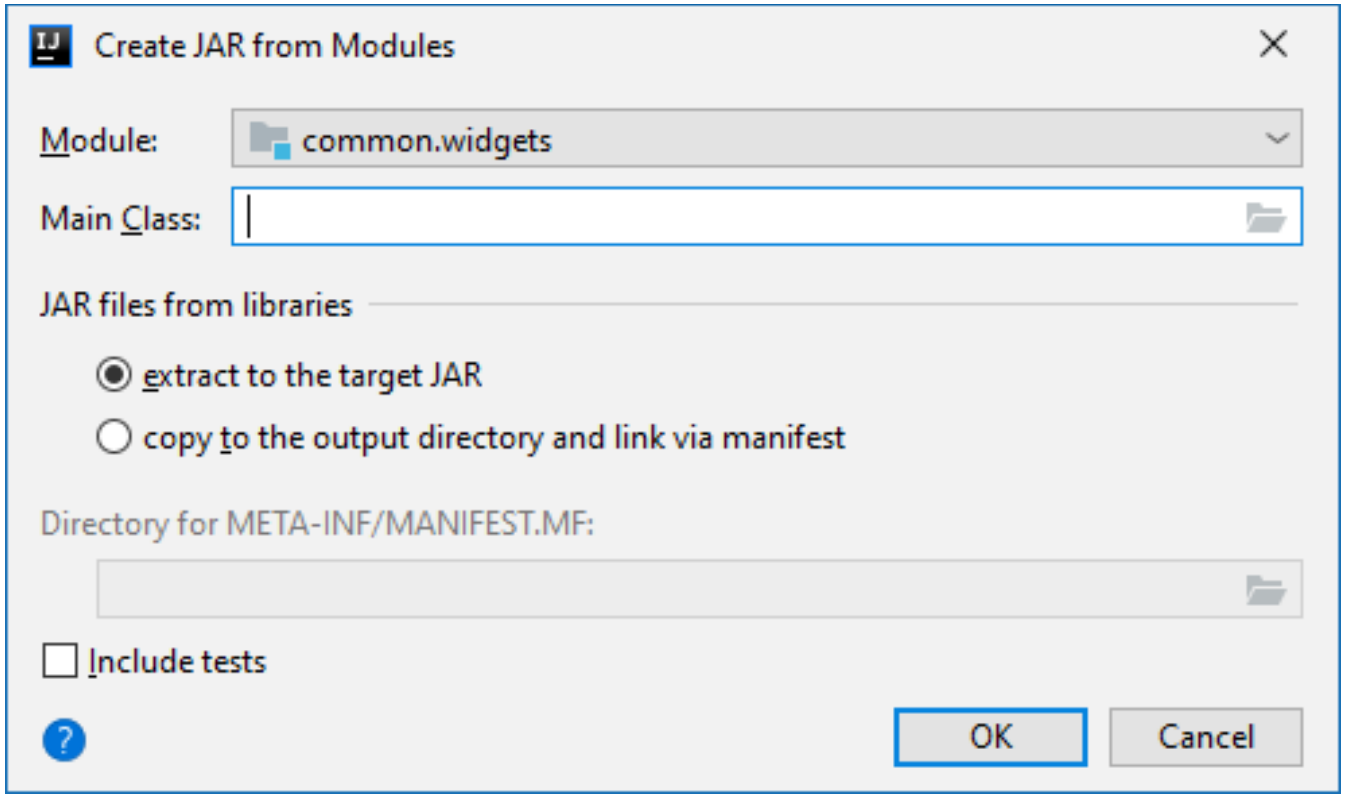
1. Open the **Project structure** and click artifacts from the left pane.
2. In the middle pane, click the + sign and the **Add** menu opens; select **JAR** and then chose from modules with dependencies.

A dialog will open like this:



6a.7 Add modules to a modular JAR

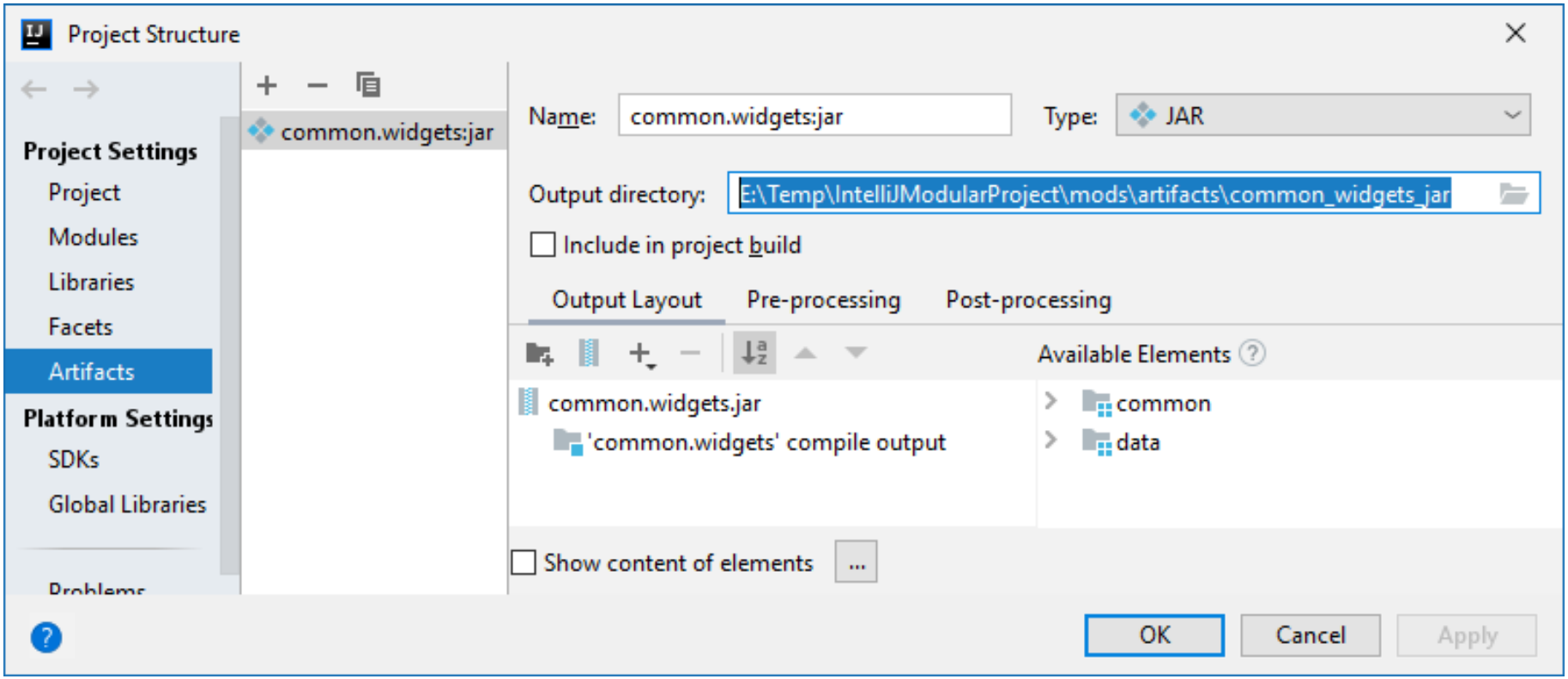
3. **Click OK.** (In case a module provides a class with a main() method, you can select it in the **Main Class** textbox)



Make sure that each module JAR only contains its output module folder:

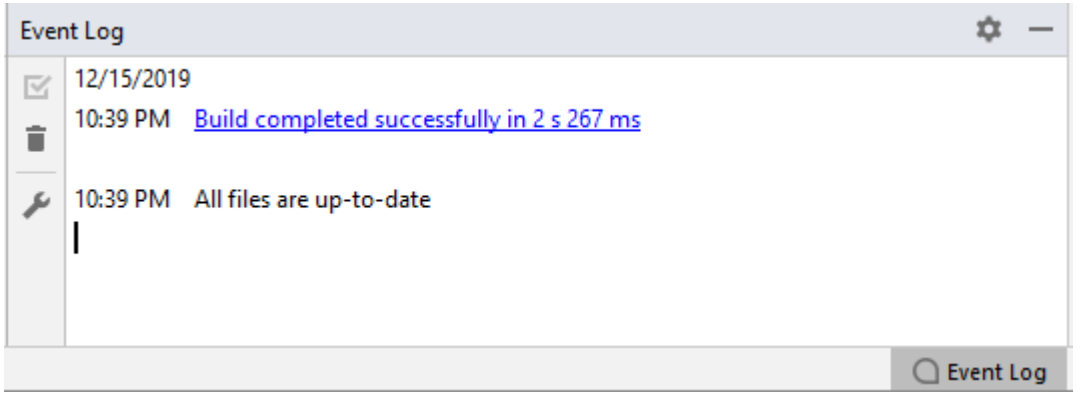
6a.7 Add modules to a modular JAR

4. Make sure that the module JAR only contains its output module folder:



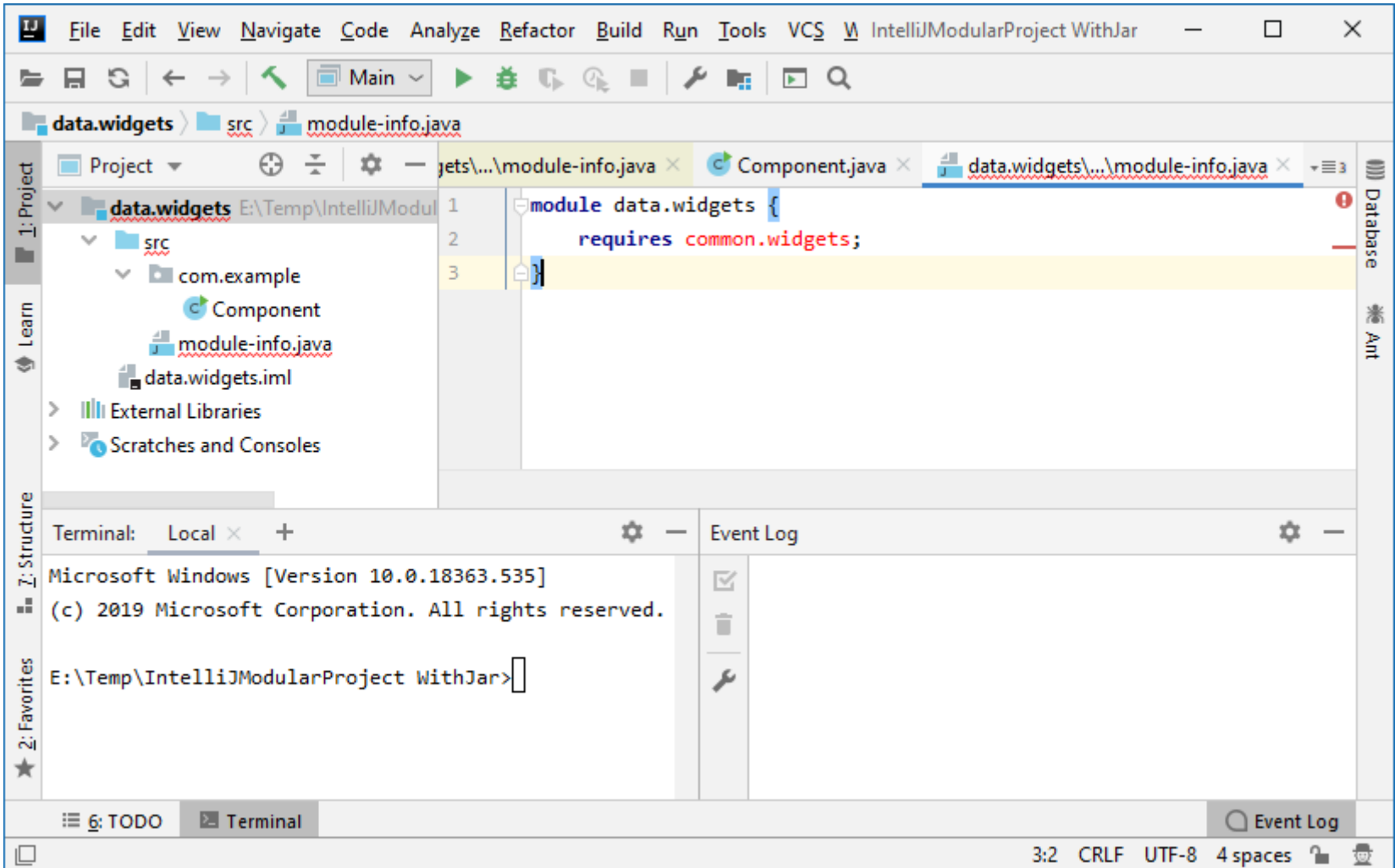
6a.7 Add modules to a modular JAR

5. Click OK to **close the Project structure** dialog. The final step is to package the modules.
6. From the main menu, choose Build, and under the **Build** folder, click **Build artifacts**
7. From the menu, select each artifact you want to build; you should see the message "*Compilation completed successfully in … ms*" in the **Event log** window



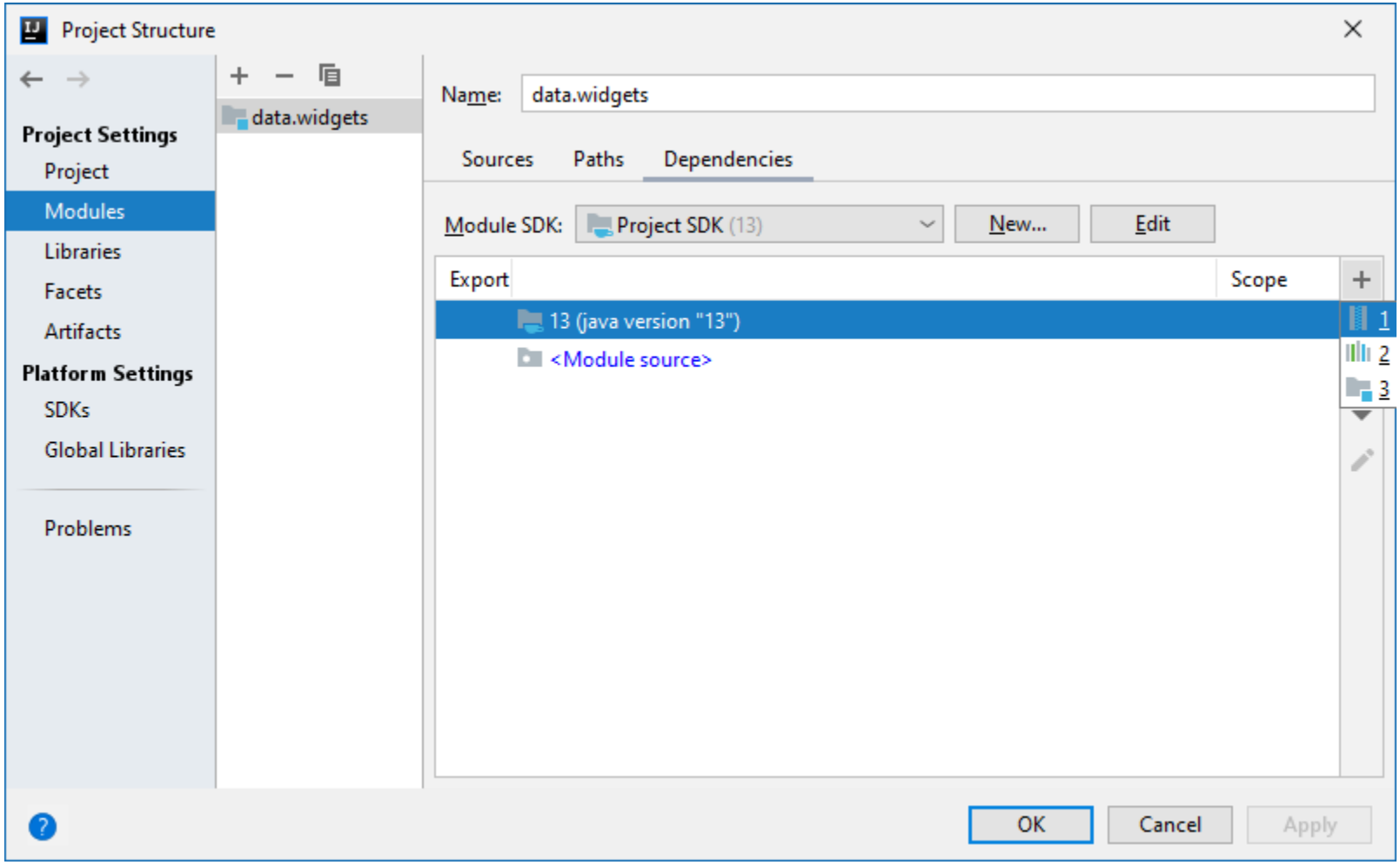
6a.7 Add modules to a modular JAR

8. Create a Project that requires the module in the JAR



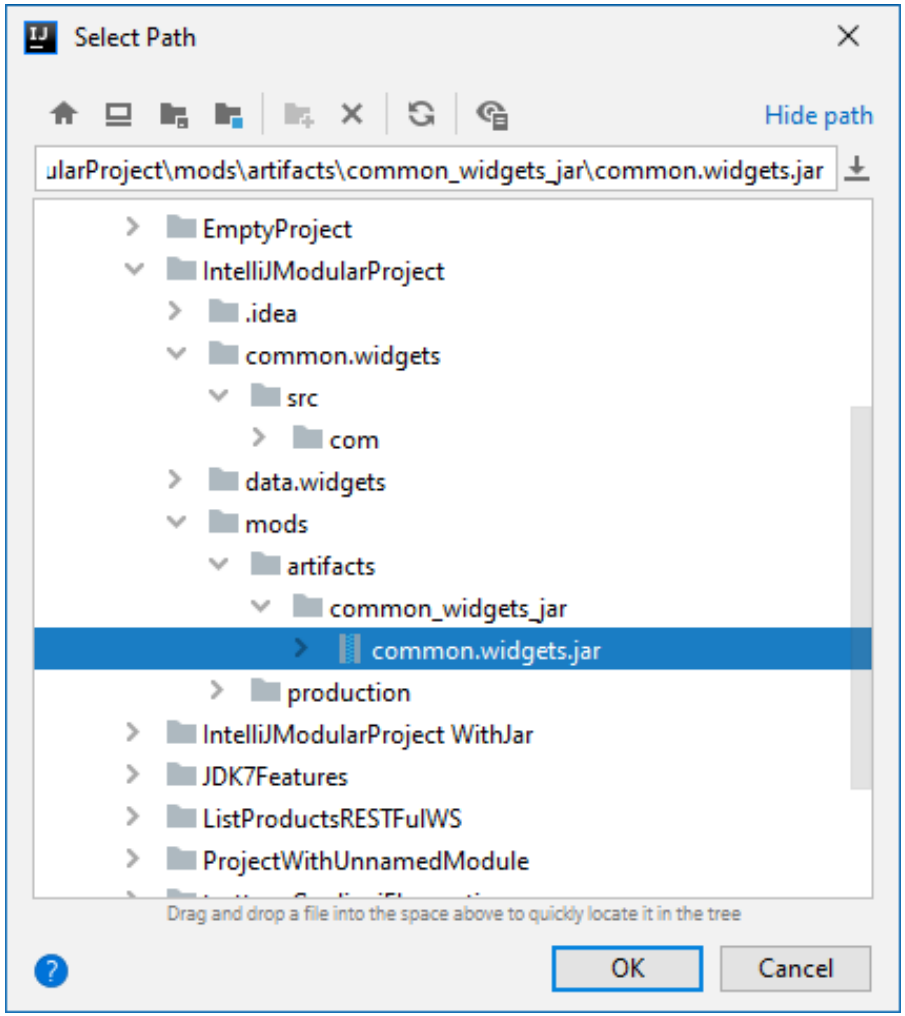
6a.7 Add modules to a modular JAR

9. Add the JAR with the required module
   (1 Select 1 JARs or directories)



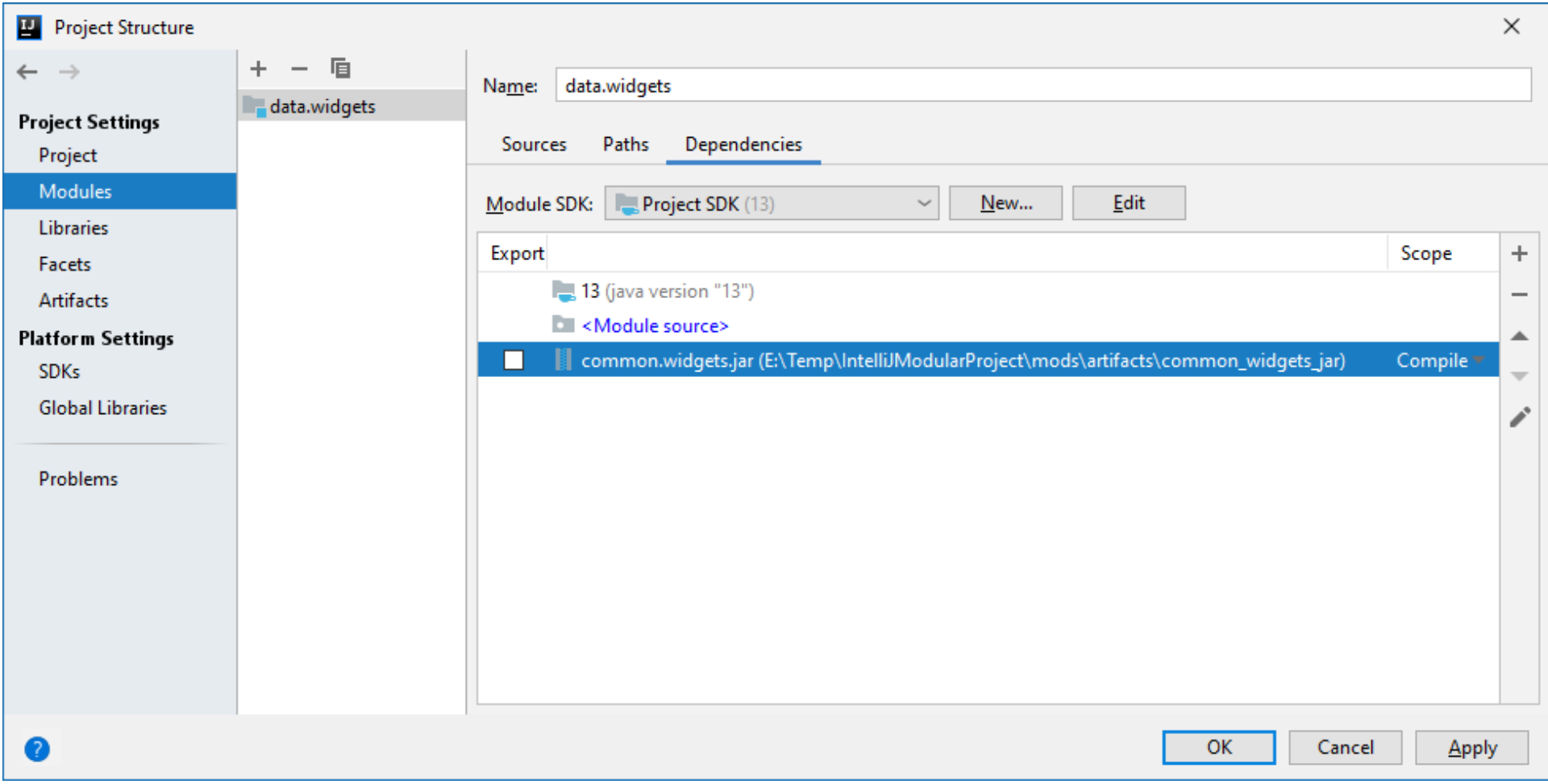
6a.7 Add modules to a modular JAR

10. Locate the Module JAR in its output module folder



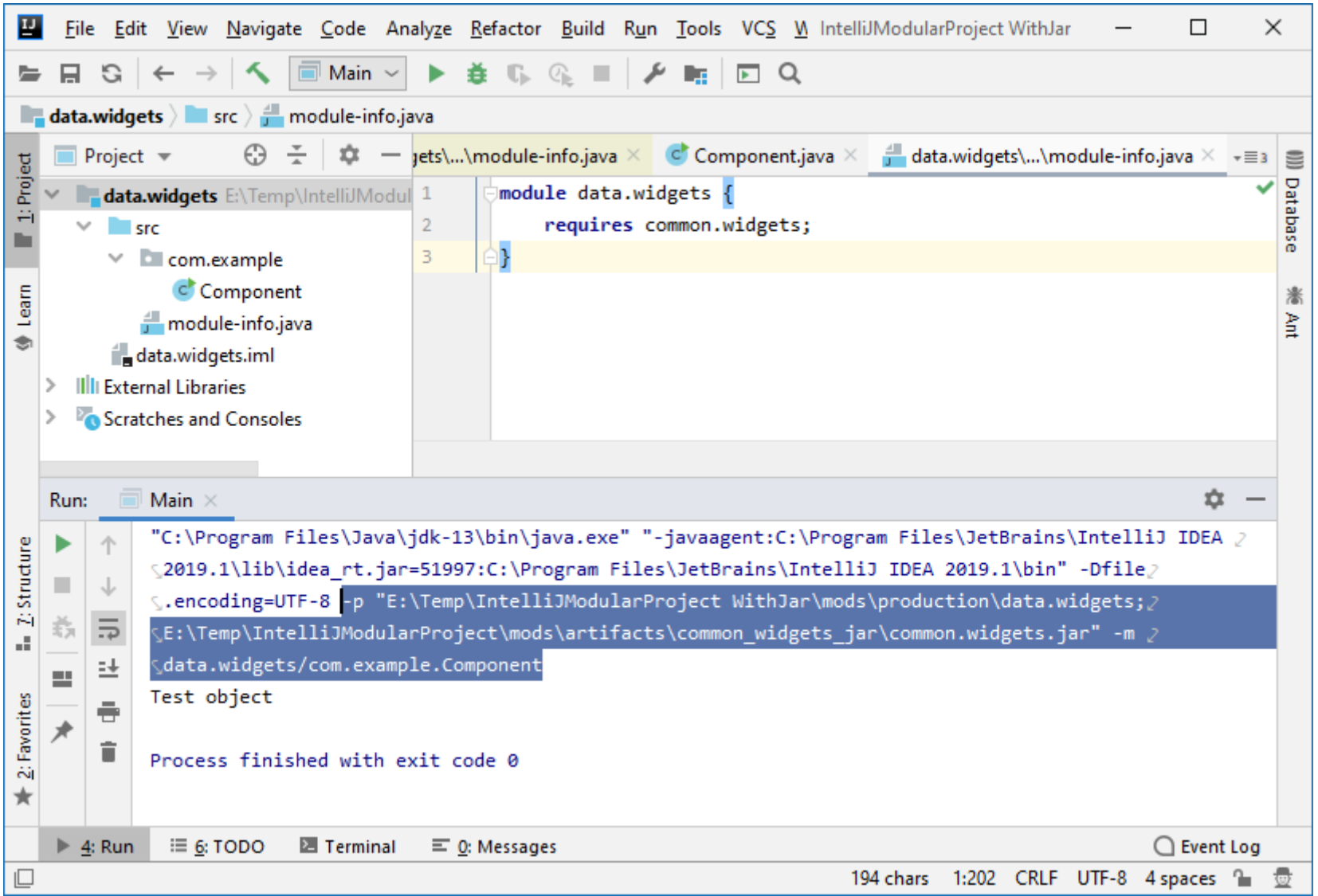
6a.7 Add modules to a modular JAR

10. Locate the Module JAR in its output module folder



6a.7 Add modules to a modular JAR

11. The project compiles and it is ready to Run



Problems

Create a Modular JavaFX application with the following design. On clicking any button display an Alert message box displaying the text written on the button. On closing the Alert message box add the same text in the TextArea