Java 9 Modules

vs. NetBeans Modules

Ioannis Kostaras

NetBeans Day Athens

26 August 2016

Agenda

* The need for a modular API
* Java 9 Modules
* NetBeans 9 EA support for JDK 9 EA
* NetBeans RCP Module API
* Java 9 Modules vs NetBeans Module API
* Recap

Java 9 Features

* jshell (project Kulla) a Read-Eval-Print-Loop (REPL) command line tool (JEP 222)
* Benchmarking Java Microbenchmarking Harness (JMH) (JEP 230)
* An HTTP 2.0 Client for HTTP 2.0 and WebSockets (JEP 110).
* Process API Improvements to improve the API for controlling and managing OS processes (JEP 102).
* Improved contended locking for increasing performance between threads (JEP 143).
* Segmented Code Cache to improve execution time for complicated benchmarks (JEP 197).
* Smart Java Compilation (Part 2) makes the `sjavac` tool available in the JDK (JEP 199).
* Modular Source Code organizes JDK source code into modules (JEP 201).

Modularisation & Modular Architecture

* *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.
* A modular application, in contrast to a monolithic one of tightly coupled code in which every unit may interface directly with any other, is composed of smaller, separated chunks of code that are well isolated.
* Versioning: depend on a specific or a minimum version of a module

Modularisation & Modular Architecture (cont.)

* *Characteristics of modular systems:*
  * **Strong encapsulation**: A module must be able to conceal part of its code from other modules. Consequently, encapsulated code may change freely without affecting users of the module.
  * **Well-defined interfaces**: modules should expose well-defined and stable interfaces to other modules.
  * **Explicit dependencies**: dependencies must be part of the module definition, in order for modules to be self-contained. A *module graph*: *nodes* represent *modules*, and *edges* represent *dependencies* between modules

Pre Java 9

Packages & Access modifiers

* Classes are arranged into packages
  * `com.company.app.MyClass` → `com/company/app/MyClass.java`
* Packages are globally visible and open for extension
* Unit of delivery is a Java archive (jar)
  * Access control is only managed in the level of classes/methods
* Classes and methods can restrict access by these access modifiers:
  * `public`
  * `protected`
  * `private`

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

Packages & Access modifiers

- How do you access a class from another package, but preventing other classes from using it?
  - You can only make the class `public`, thus exposing it to all other classes ➔ **breaks encapsulation**
- No explicit dependencies
  - explicit import statements are only at compile time; there is no way to know which other JAR files your JAR needs at run-time; user has to provide correct jars in classpath during execution

  ➔ Maven or OSGi
  - Maven solves compile-time dependency management by defining POM (Project Object Model) files. (Gradle works in a similar way)
  - OSGi solves run-time dependencies by requiring imported packages to be listed as metadata in JARs, which are then called bundles

Classpath

* Once a classpath is loaded by the JVM, 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 happens when duplicate classes are in the classpath? ➔ Only one wins
* The JVM cannot efficiently verify the completeness of the classpath upon starting. If a class cannot be found in the classpath, then you get a run-time exception.
* The term "Classpath Hell" or "JAR Hell" should now be clearer to you
*

Java 9 Modules
Project Jigsaw

Java 9 Modules Goals

* Java Platform Module System (JSR 376)
  * Reference implementation: OpenJDK Project Jigsaw
* Modular JDK (JEP 200)
* Modularize the layout of the source code in the JDK (JEP 201).
* Modularize the structure of the binary runtime images (JEP 220).
* Disentangle the complex implementation dependencies between JDK packages.

Java 9 Module System

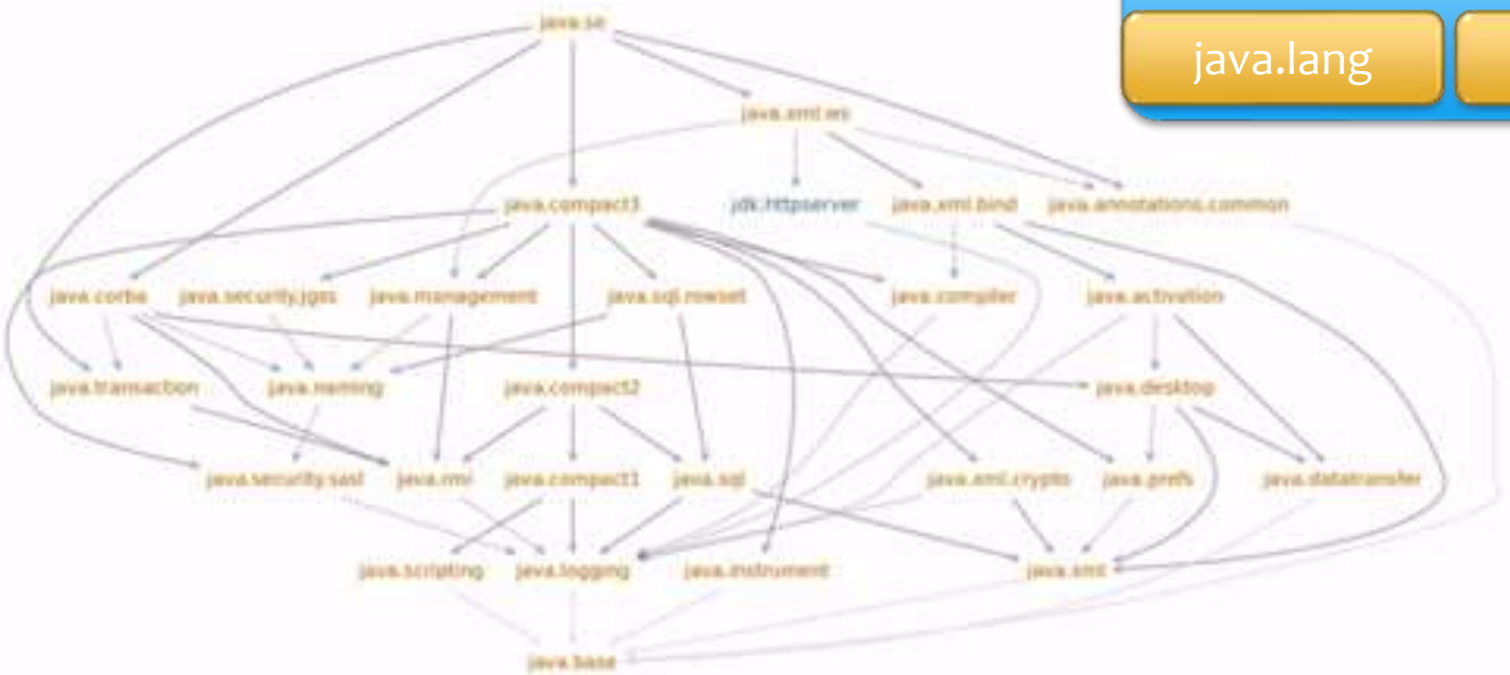
* Modules can either export or strongly encapsulate packages
* Modules express dependencies on other modules explicitly.
* Each JAR becomes a module, containing explicit references to other modules.
* A module has a publicly accessible part and an encapsulated part.
* All this information available at compile-time and run-time
* Accidental dependencies on code from other non-referenced modules can be prevented.
* optimizations can be applied by inspecting (transitive) dependencies

Benefits of Java 9 Module System

* **Reliable configuration:** The module system checks whether a given combination of modules satisfies all dependencies before compiling or running code
* **Strong encapsulation:** Modules express dependencies on other modules explicitly.
* **Scalable development:** Teams can work in parallel by creating explicit boundaries that are enforced by the module system.
* **Security:** No access to internal classes of the JVM (like Unsafe).
* **Optimisation:** optimizations can be applied by inspecting (transitive) dependencies. It also opens up the possibility to create a minimal configuration of modules for distribution

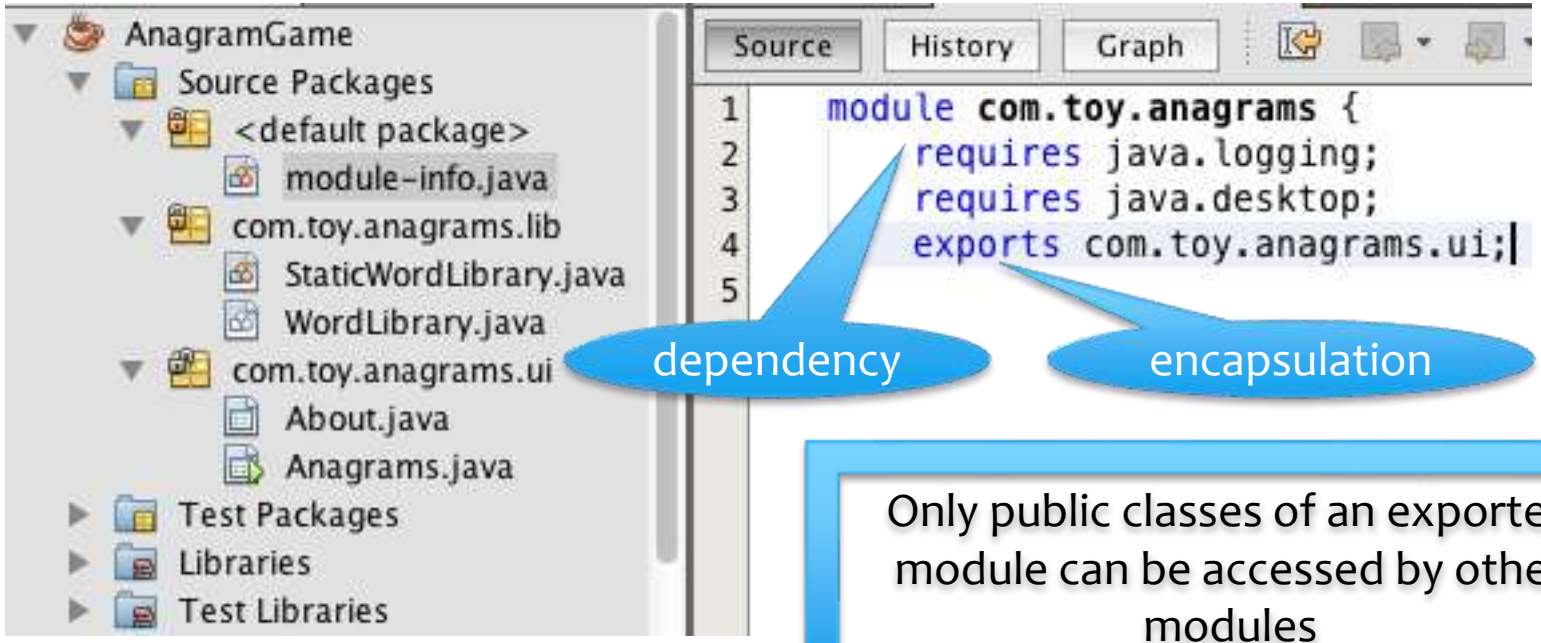
JDK 9 Platform Modules

* Module `java.base` exposes packages `java.lang`, `java.util` etc. It is the core Java module which is imported by default
* JDK now consists of about 90 platform modules



Modules in Java 9

* A module has a name (e.g. `java.base`), it groups related code and possibly other resources, and is described by a module descriptor.
* Like packages are defined in `package-info.java`, modules are defined in `module-info.java` (in root package)
* A *modular* jar is a jar with a `module-info.class` inside it



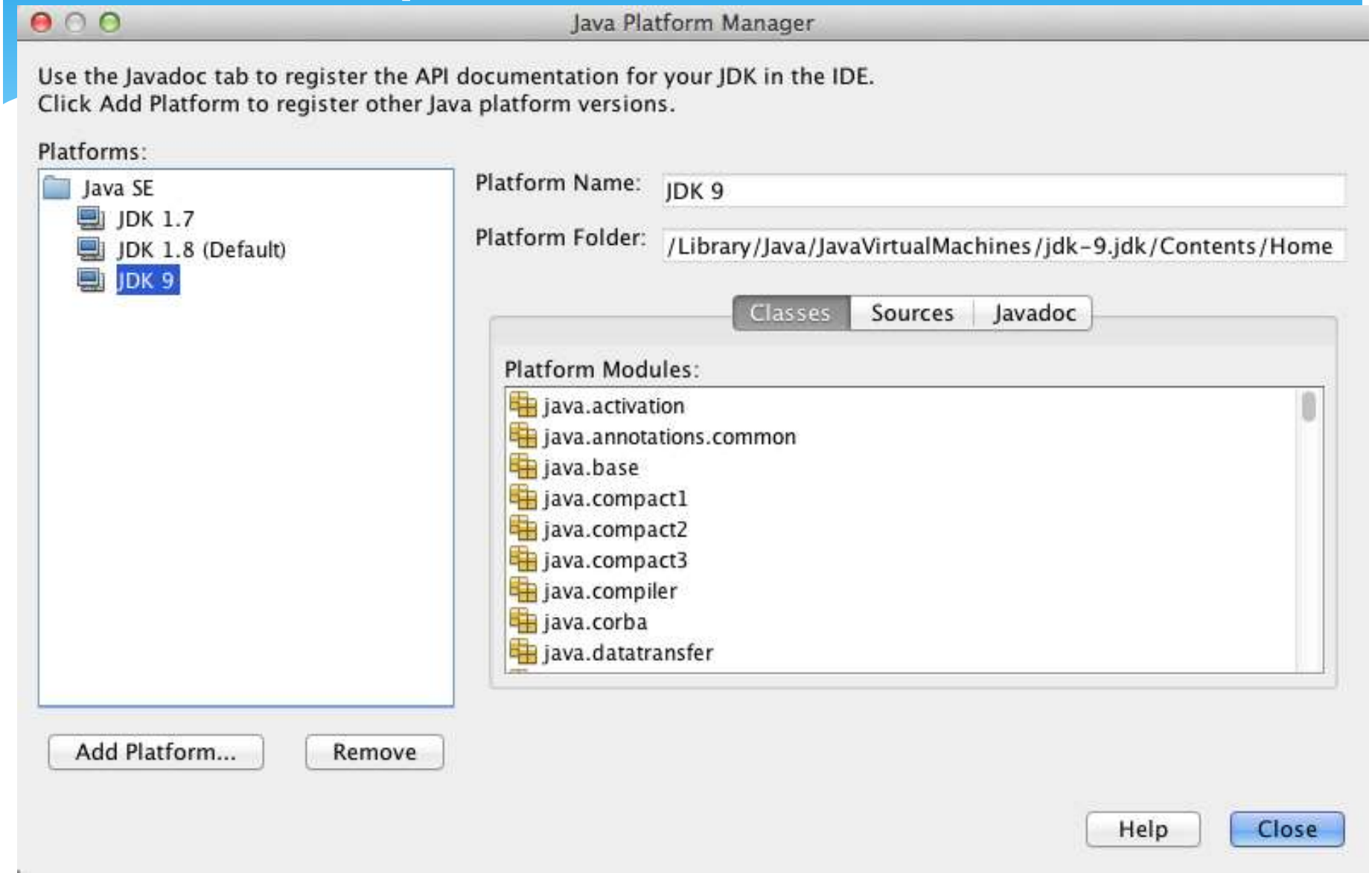
Only public classes of an exported module can be accessed by other modules

NetBeans 9 EA

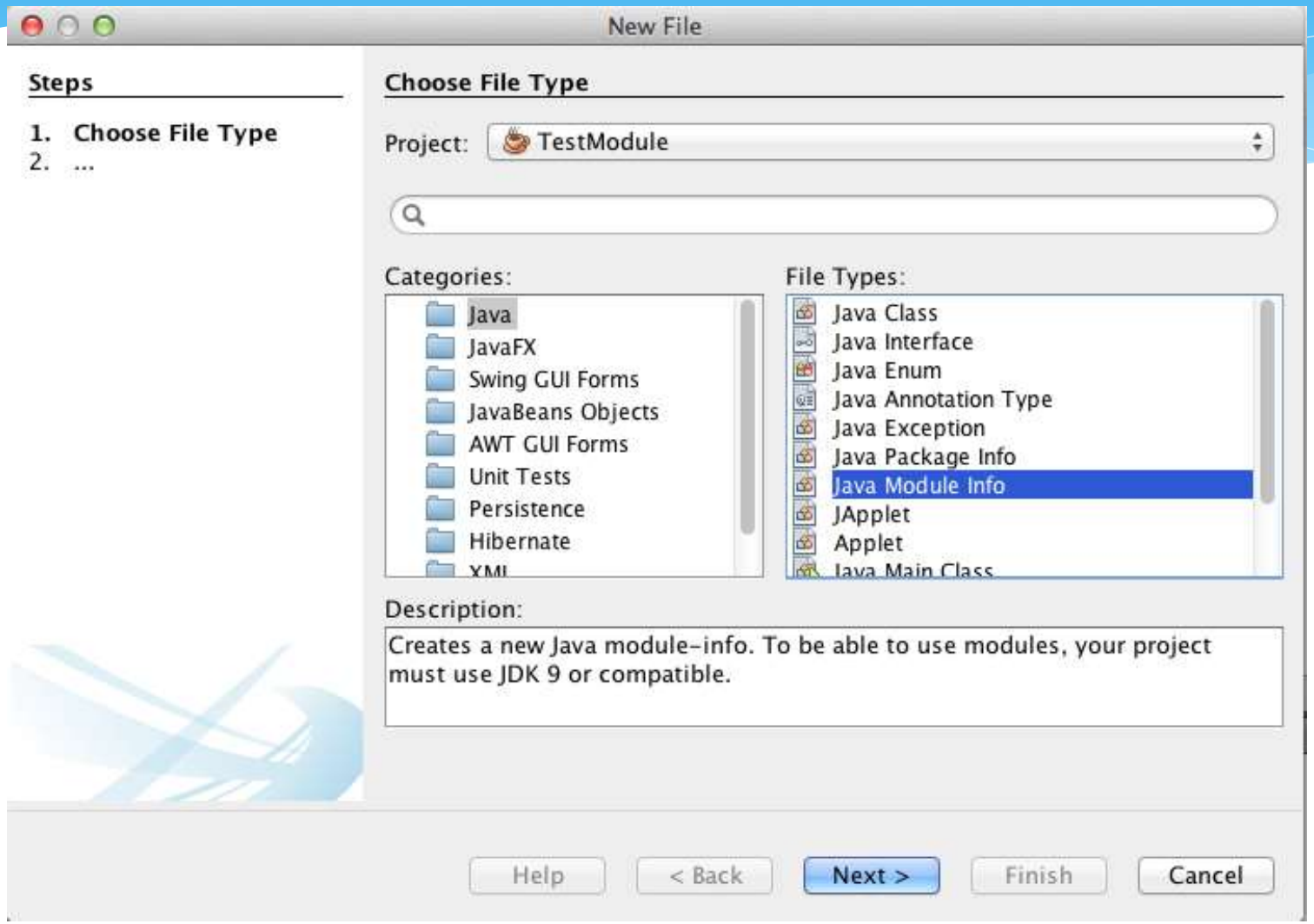
Getting Started

* Download the latest JDK 9 Early Access from https://jdk9.java.net/download/ page, build 111 or newer.
* Download NB JDK 9 dev build from http://bits.netbeans.org/netbeans/nb9-for-jdk9_jigsaw/daily/latest/ or build it from sources.
* Configure it to run with JDK 8 (`etc/netbeans.conf`)
* Register the latest JDK 9 EA build as a *Java Platform* in NetBeans by means of **Tools →Java Platforms → Add Platform.**

Setup JDK 9 EA Platform



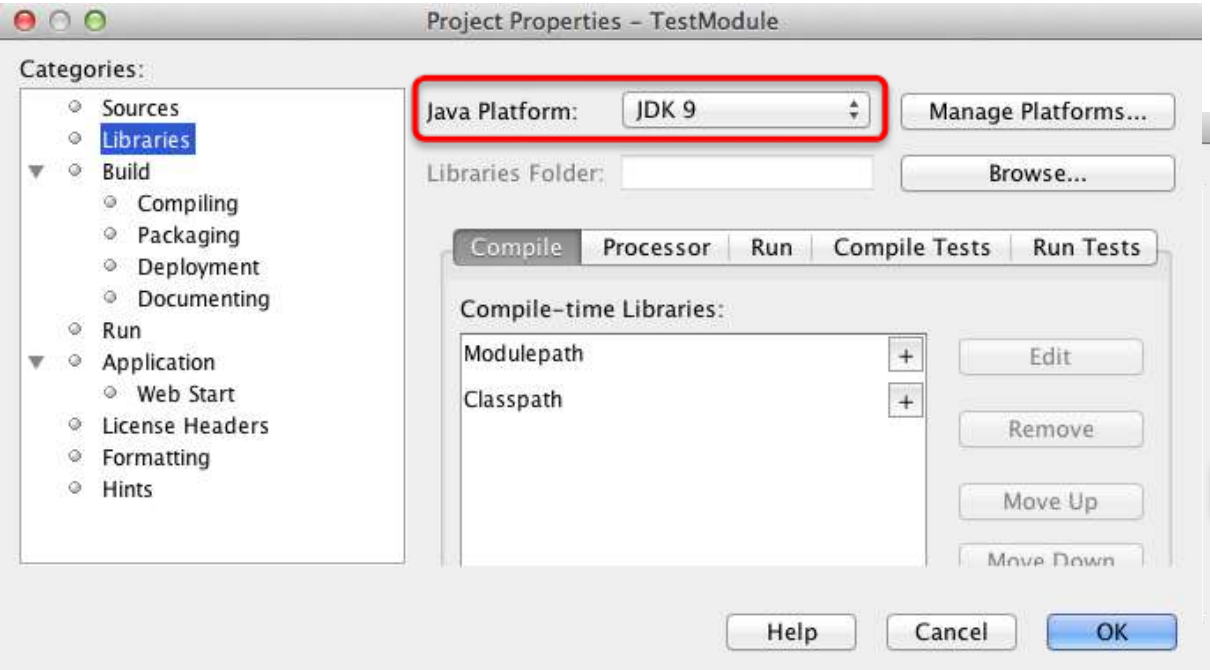
Add module-info.java to a Java Project

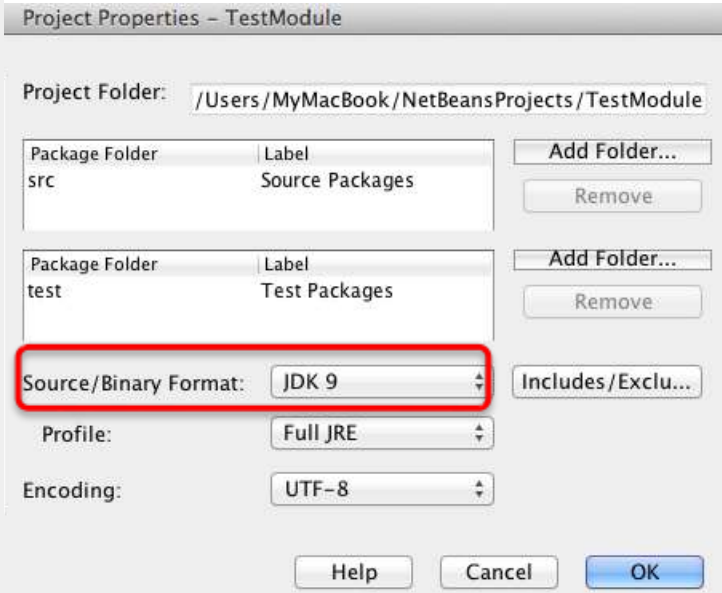


Set JDK 9 to project level

Setup the project to JDK9 in project Properties:

* In *Libraries* set **Java Platform** to your JDK 9 EA Java platform.
* In *Sources* set **Source /Binary Format** to **JDK 9**





module-info.java

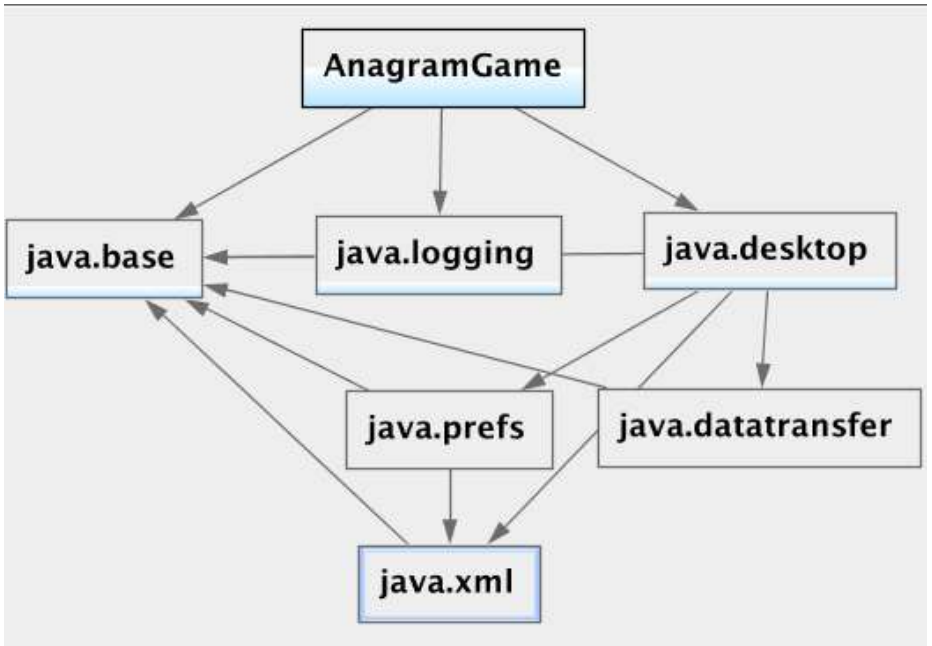
Projects – Todo

- AnagramGame
  - Source Packages
    - <default package>
      - module-info.java
    - com.toy.anagrams.lib
      - StaticWordLibrary.java
      - WordLibrary.java
    - com.toy.anagrams.ui
      - About.java
      - Anagrams.java
  - Test Packages
  - Libraries
  - Test Libraries

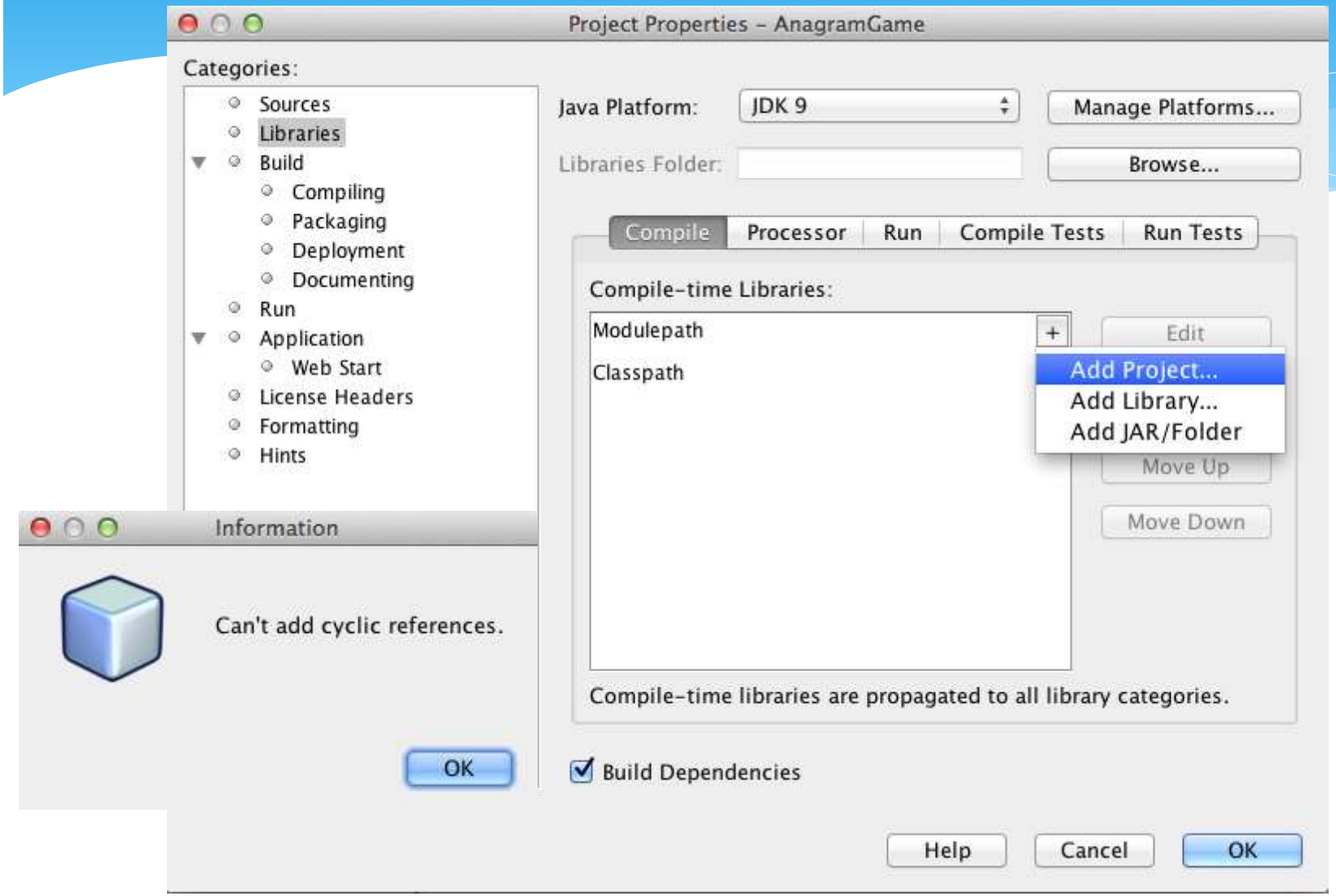
Start Page | module-info.java

Source | History | Graph

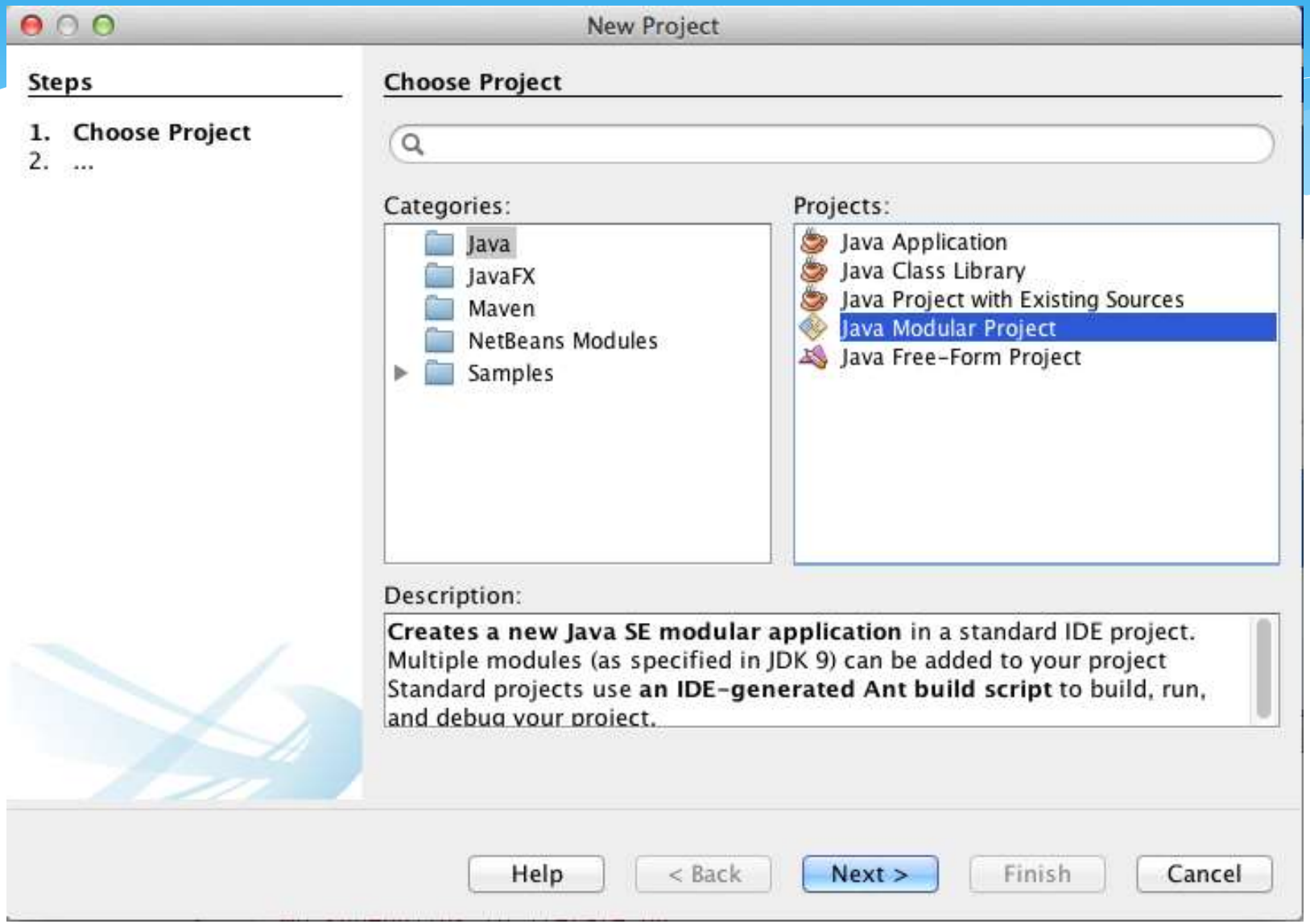
```
1  module AnagramGame {
2      requires java.logging;
3      requires java.desktop;
4  }
5
```



Module dependencies



New Java Modular Project



NetBeans 9 EA

* Multiple-module project support?
  * Multiple JDK9 modules in one NB project
  * Maven projects don't work
* NetBeans 9 EA doesn’t seem to allow `module-info.java` to be in another package than in root package (for multiple-module projects)
* Export/hide a package from a popup menu entry?
* Provide support for locating a module that contains a specific package (to include in `module-info.java`)
  * `java --list-modules <package name>`
* How to create modular runtime images from NetBeans (see `jlink`)?
* Display error when try to import an internal library/package (e.g. `sun.invoke.util.BytecodeName`)
* Support for jshell?

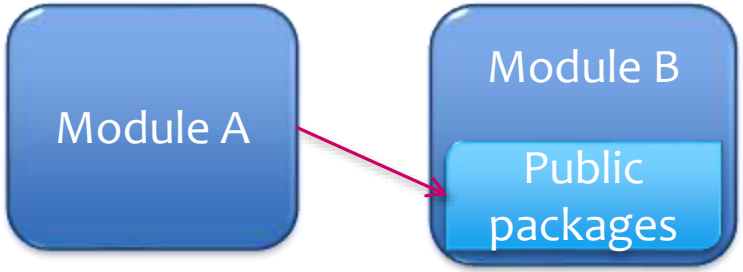
NetBeans Module API

NetBeans Module API overview

* NetBeans Module API
  * is an architectural framework
  * is an execution environment that supports a module system called *Runtime Container*.
* The Runtime Container consists of the minimum modules required to load and execute your application.

Modules

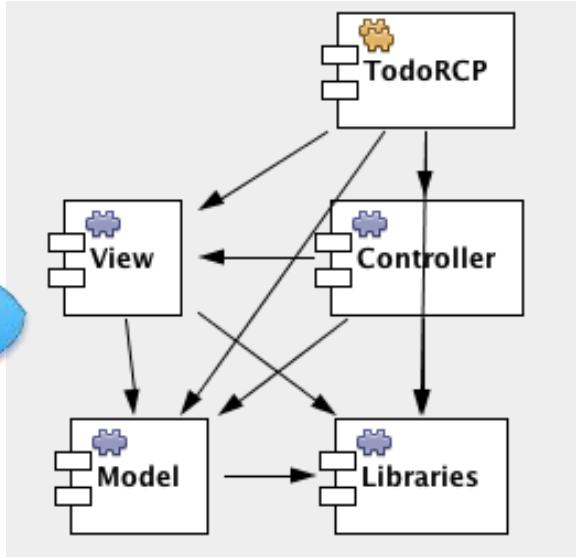
* A ***module*** is a collection of functionally related classes stored in a JAR file along with metadata, which provide information to the Runtime Container about the module, such as
  * the module's name,
  * version information,
  * dependencies, and
  * a list of its public packages, if any.

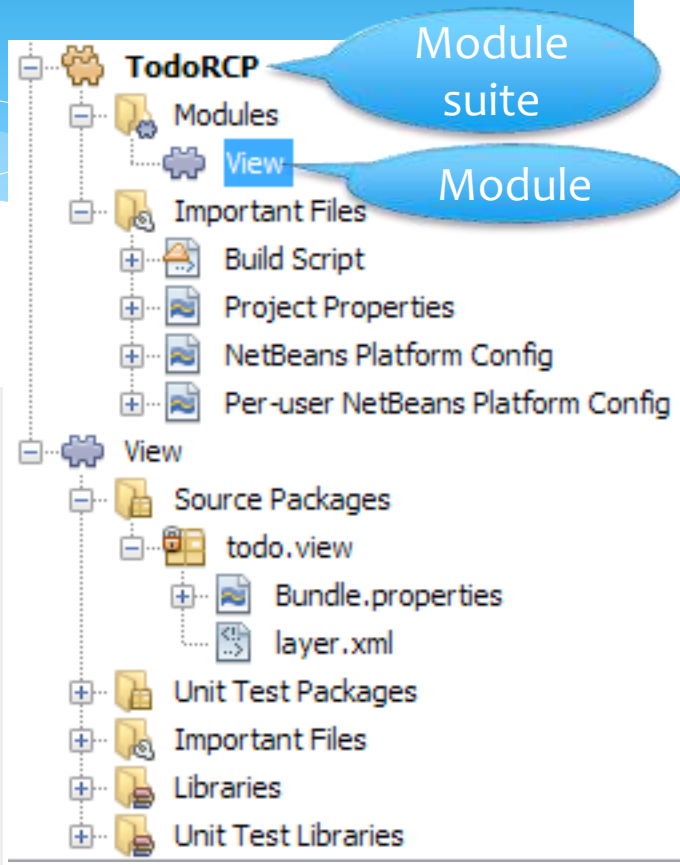


* In order to use or access code in another module:
  1. You must put Module B classes in a public package and assign a version number.
  2. Module A must declare a dependency on a specified version of Module B.

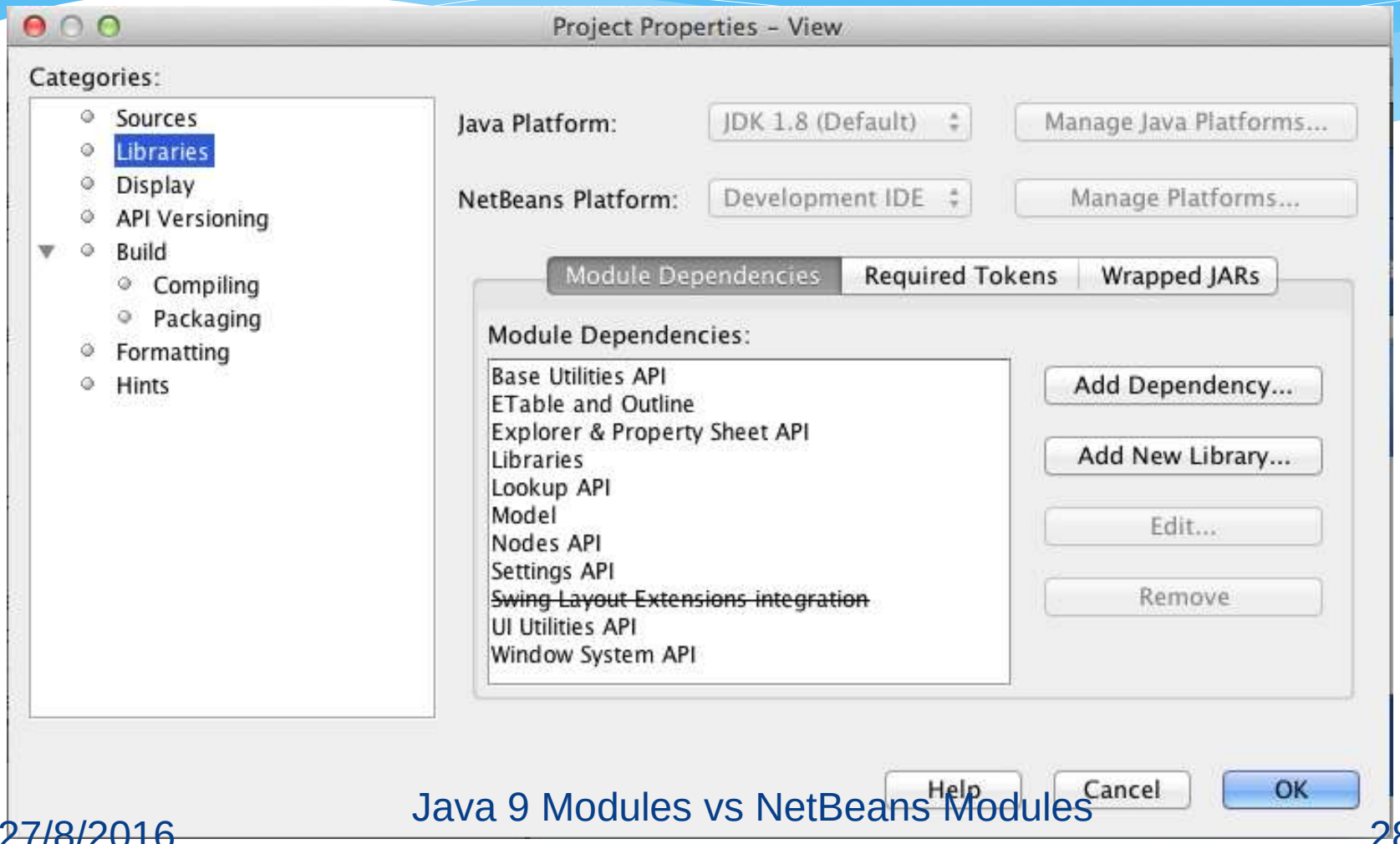
Modules and Module Suites

1. **File → New Project → NetBeans Modules → NetBeans Platform Application** creates a suite of modules
2. **Right-click on Modules → Add New**
   * View (todo.view)

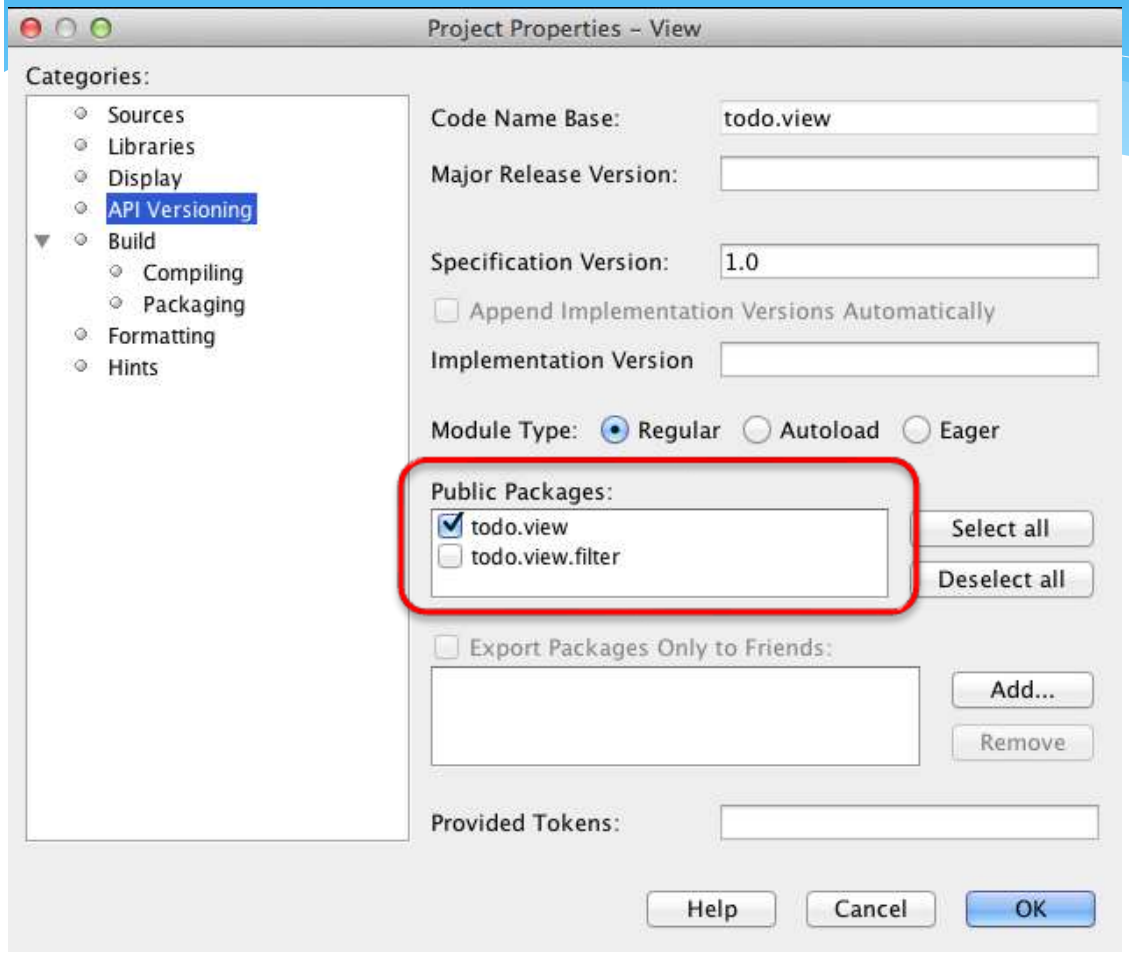


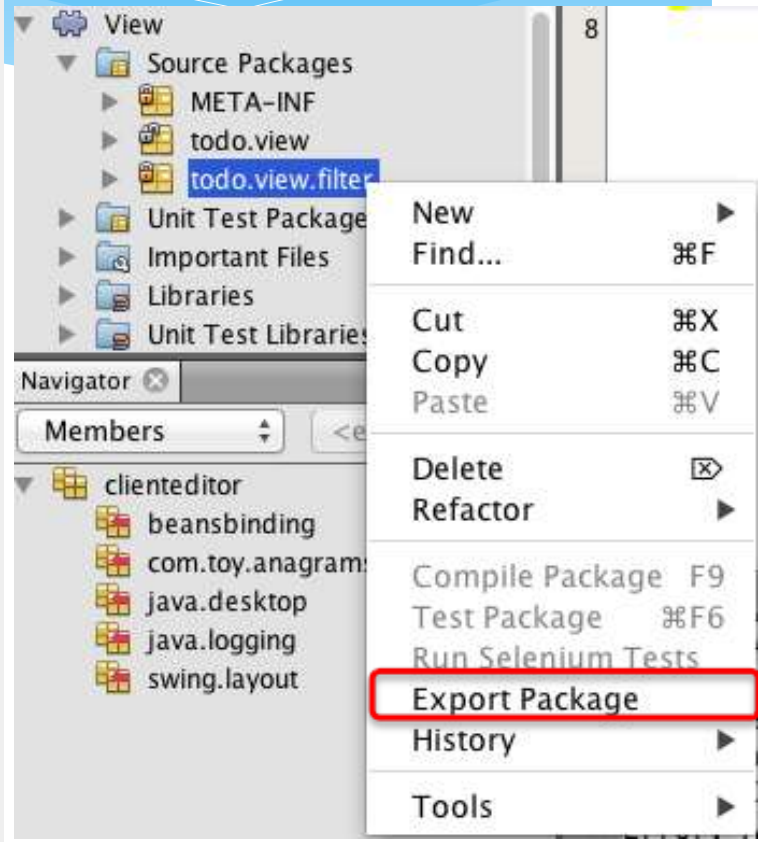


Module Dependencies



Public packages





Interfaces

* NetBeans RCP provides a `@ServiceProvider` that allows for loose coupling between modules.

```
@ServiceProvider(service = Provider.class, position=1)
public class ProviderImpl implements Provider { }
```

* A lookup is a map with class objects as keys and sets of instances of these class objects as values, i.e. `Lookup = Map<Class, Set<Class>>`, e.g. `Map<String, Set<String>>` or `Map<Provider, Set<Provider>>`. NetBeans provides a number of methods to access a lookup:

```
Provider provider =
    Lookup.getDefault().lookup(Provider.class);
provider.aMethod();
```

* or if you have more than one implementations of Provider:

```
Collection <? extends Provider> providers =
    Lookup.getDefault().lookupAll(Provider.class);
for (Provider provider : providers) { }
```

Java 9 Modules
vs
NetBeans Modules

Java 9 modules vs NB Modules

	Java 9 Modules	NetBeans Module API
Encapsulation	✓	✓
Interfaces	✓	✓
Explicit dependencies	✓	✓
Versioning	✗	✓
Cyclic dependencies*	✗	✗

Recap

* Java 9 introduces a module system (project jigsaw)
* NetBeans 9 EA provides support for JDK 9 EA (project jigsaw)
* NetBeans RCP has its own Module API based on OSGi
* Comparison of NetBeans Module API to the Java 9 Module System API

References

* Evans, B. (2016), "An Early Look at Java 9 Modules", *Java Magazine*, Issue 27, pp.59-64.
* Mak S. & Bakker P. (2016), *Java 9 Modularity*, O'Reilly (Early Release)
* Reinhold M. & Bateman A. (2015), "*Introduction to Modular Development*", Devoxx.
* Reinhold M. & Bateman A. (2015), "*Advanced Modular Development*", Devoxx.
* NetBeans 9 EA Support

Questions