Agenda

- Overview of Java 9 Features
- 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

Java 9 is Feature Complete!

- Modularity
  - 200: The Modular JDK (Jigsaw/JSR 376 and JEP 261)
  - 201: Modular Source Code
  - 220: Modular Run-Time Images
  - 238: Multi-Release JAR Files
  - 261: Module System
  - 275: Modular Java Application Packaging
  - 282: jlink: The Java Linker

Java 9 Features

Java 9 is Feature Complete!

* Developer Convenience
  * 193: Variable Handles
  * 213: Milling Project Coin
  * 222: jshell: The Java Shell (Read-Eval-Print Loop) - (project Kulla)
  * 259: Stack-Walking API
  * 266: More Concurrency Updates to `CompletableFuture` and support for Reactive Streams.
  * 269: Convenience Factory Methods for Collections
  * 276: Dynalink
  * 277: Enhanced Deprecation
  * 285: Spin-Wait Hints

Java 9 Features

Java 9 is Feature Complete!

- Strings
  - 250: Store Interned Strings in CDS Archives
  - 280: Indify String Concatenation
- Diagnostics
  - 228: Add More Diagnostic Commands
  - 240: Remove the JVM TI hprof Agent
  - 241: Remove the jhat Tool
- JVM Options
  - 197: Segmented Code Cache to improve execution time for complicated benchmarks (?)
  - 214: Remove GC Combinations Deprecated in JDK 8
  - 245: Validate JVM Command-Line Flag Arguments
  - 248: Make G1 the Default Garbage Collector
- Logging
  - 158: Unified JVM Logging
  - 264: Platform Logging API and Service
  - 271: Unified GC Logging

Java 9 Features
Java 9 is Feature Complete!

* Javadoc
  * 224: HTML5 Javadoc
  * 225: Javadoc Search
  * 254: Compact Strings
* JavaScript/HTTP
  * 110: HTTP 2 Client: An HTTP 2.0 Client for HTTP 2.0 and WebSockets (and begin replacing "the legacy HttpURLConnection API")
  * 236: Parser API for Nashorn
  * 289: Deprecate the Applet API
  * 292: Implement Selected ECMAScript 6 Features in Nashorn
* Native Platform
  * 102: Process API Updates ("Improve the API for controlling and managing operating-system processes.")
  * 272: Platform-Specific Desktop Features

Java 9 Features
Java 9 is Feature Complete!

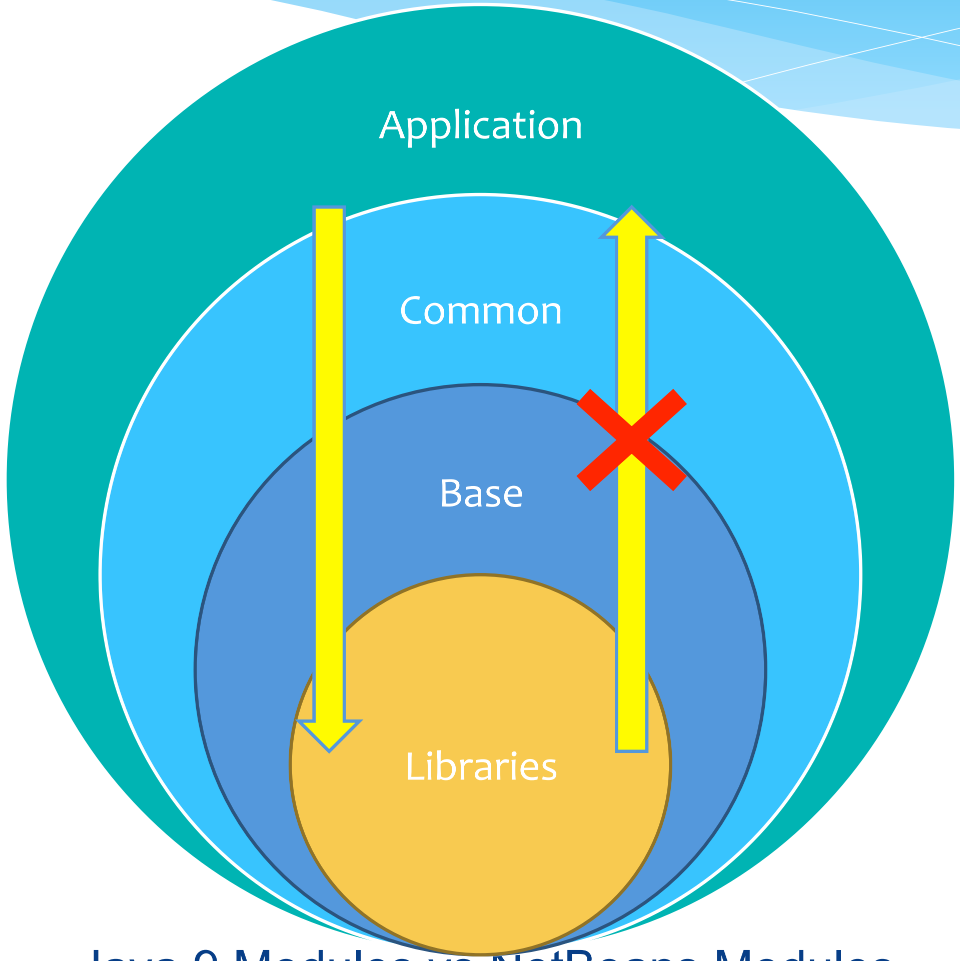
- JavaFX
  - 253: Prepare JavaFX UI Controls & CSS APIs for Modularization
  - 257: Update JavaFX/Media to Newer Version of GStreamer
- Images
  - 251: Multi-Resolution Images
  - 262: TIFF Image I/O
- Unicode
  - 227: Unicode 7.0
  - 267: Unicode 8.0
- Miscellaneous
  - 219: Datagram Transport Layer Security (DTLS)
  - 256: BeanInfo Annotations
  - 260: Encapsulate Most Internal APIs
  - 274: Enhanced Method Handles
  - 295: Ahead-of-Time Compilation

Java 9 Features ***Not In***

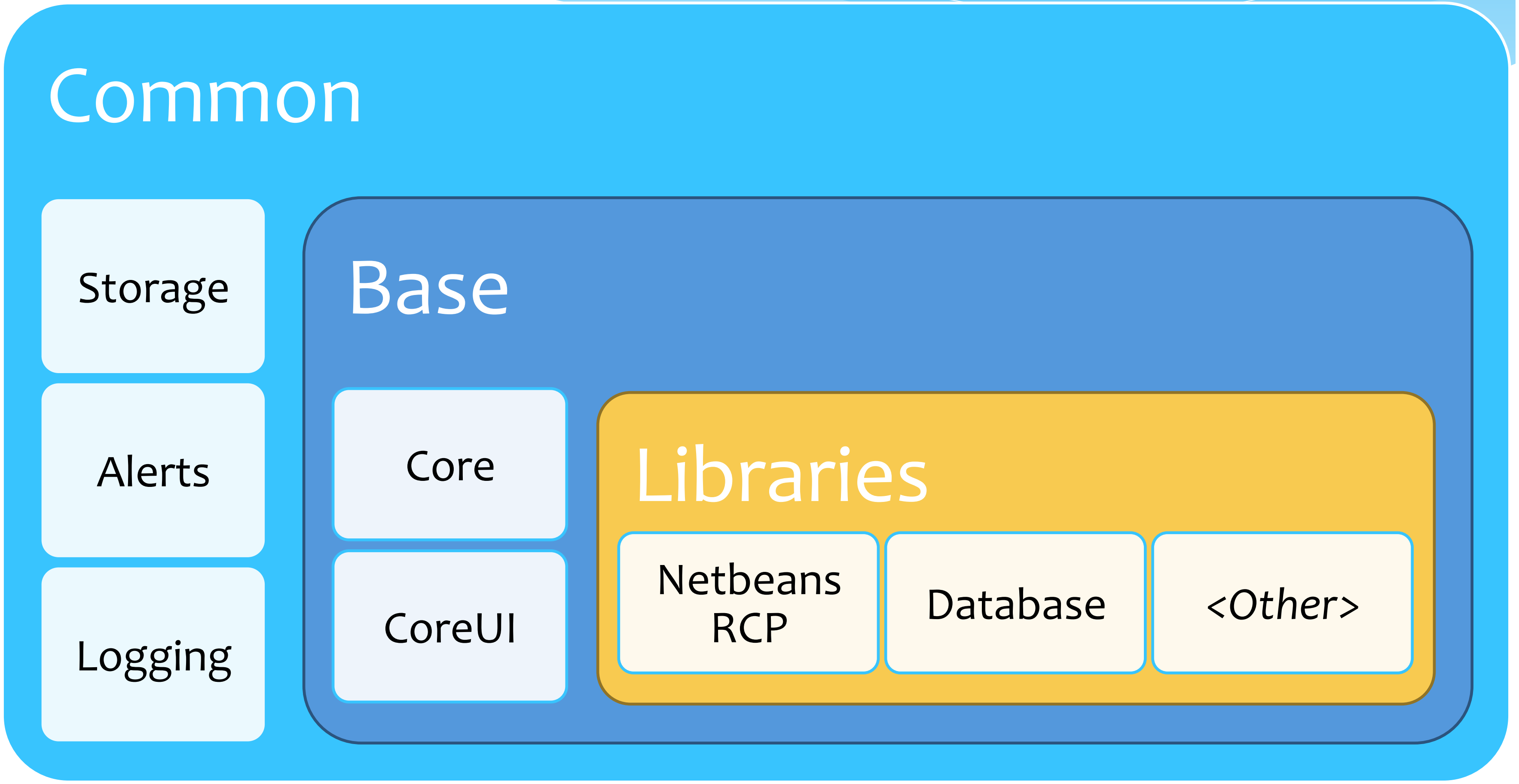
* Benchmarking Java Microbenchmarking Harness (JMH) (JEP 230)
* Smart Java Compilation (Part 2) makes the `sjavac` tool available in the JDK (JEP 199)
* Improved contended locking for increased performance between threads (JEP 143)
* Value types (JEP 169)

Modular Architecture

Modularity is one of the tools you can employ to manage and reduce complexity.



Modular Architecture



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

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

Java 9 Modules Goals

* Java Platform Module System (JSR 376)
    * Reference implementation: OpenJDK Project Jigsaw
* Module System (JEP 261)
* 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.
* Internal APIs encapsulation (JEP 260)
* Make Java SE more flexible, scalable, maintainable and secure
* Make it easier to construct, maintain, deploy and upgrade applications
* Enable improved performance

Java 9 Module System

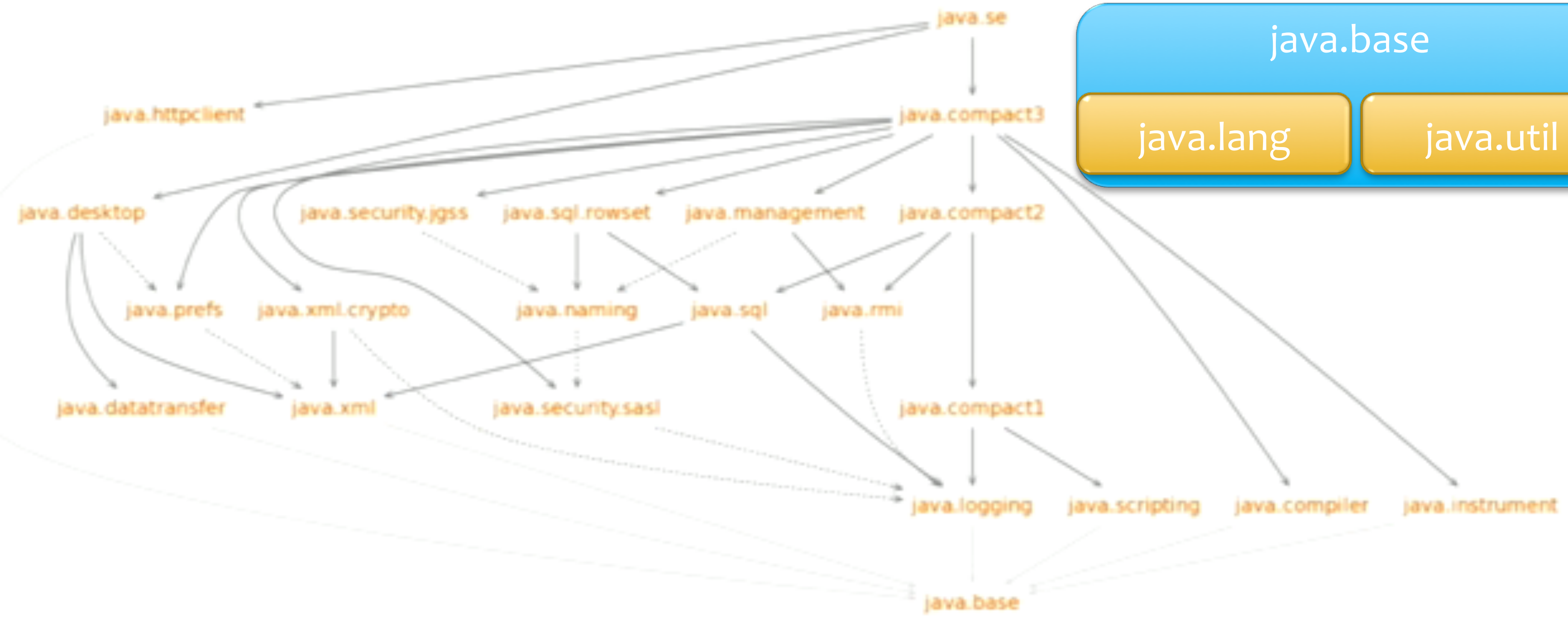
* Modules can either export or strongly encapsulate packages
* Modules express dependencies on other modules explicitly.
* Each JAR becomes a module, containing explicit references to other modules.
* A module has a publicly accessible part and an encapsulated part.
* All this information is available 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

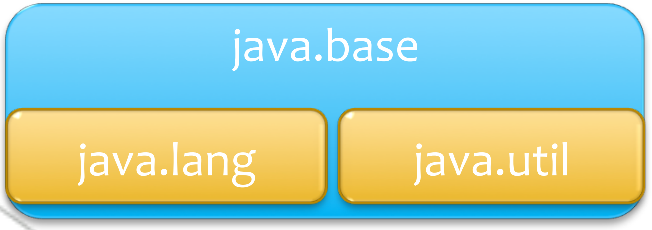
Benefits of Java 9 Module System

- **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

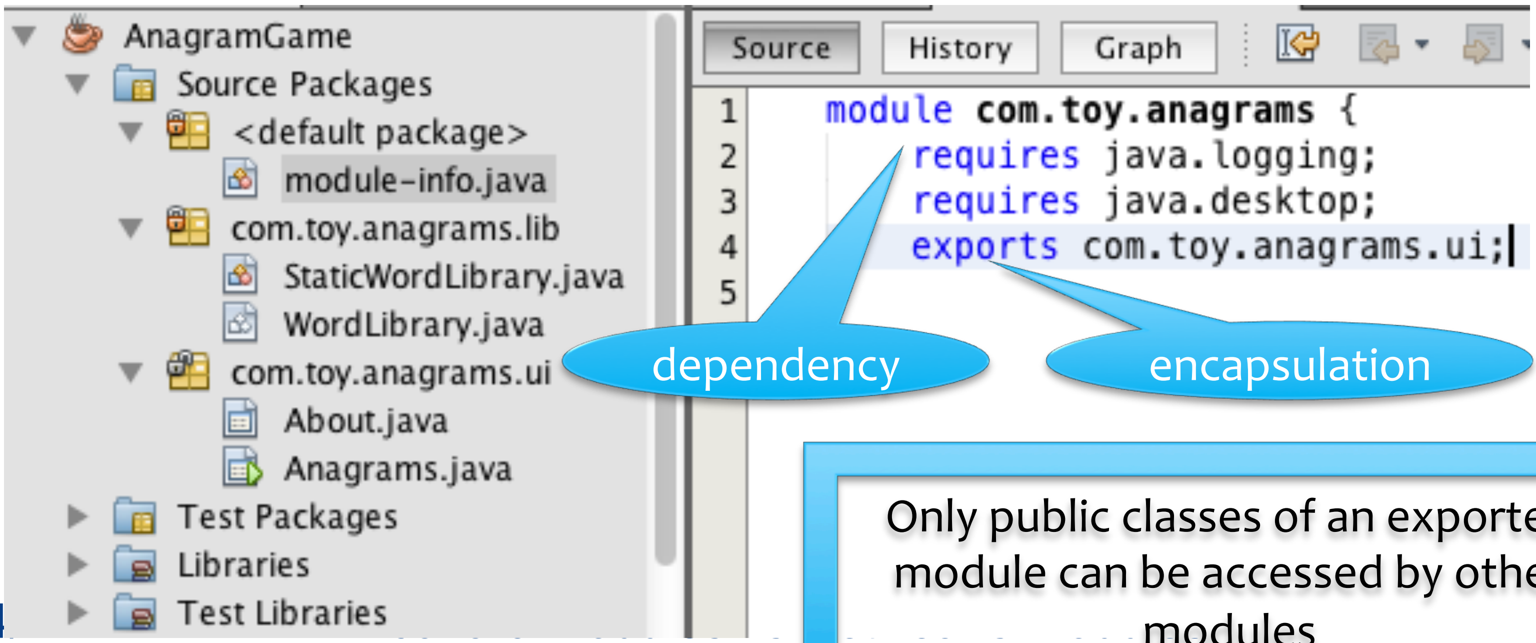
- Module `java.base` exposes packages `java.lang`, `java.util` etc. It is the core Java module which is imported by default
- JDK now consists of about 90 platform modules





Modules in Java 9

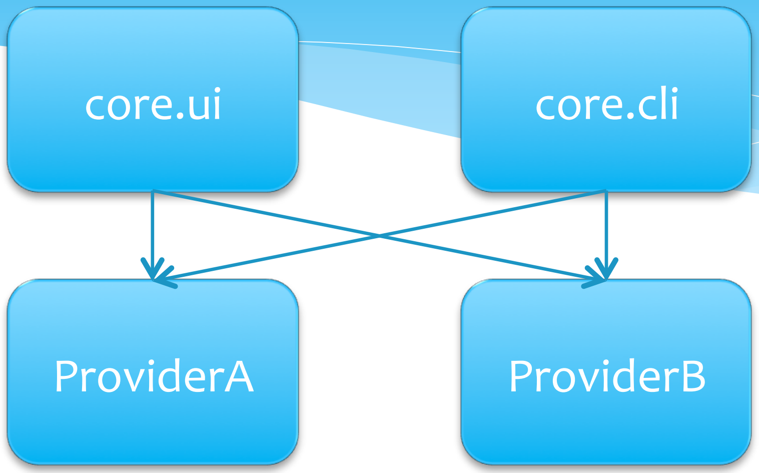
- A module has a *name* (e.g. `java.base`), it groups related code and possibly other resources, and is described by a *module descriptor*.
- Like packages are defined in `package-info.java`, modules are defined in `module-info.java` (in root package)
- A *modular* jar is a jar with a `module-info.class`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

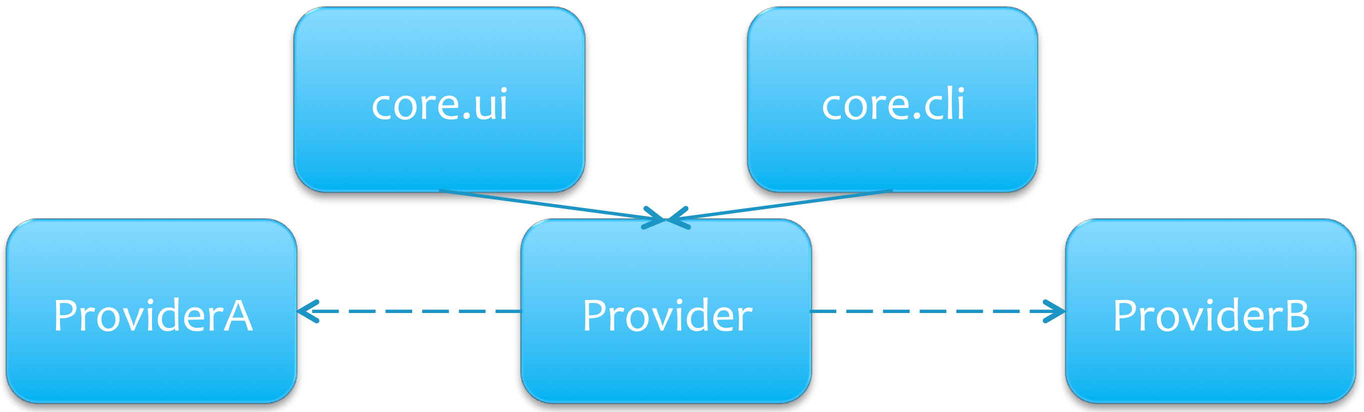


```
module com.toy.anagrams {
    requires java.logging;
    requires java.desktop;
    exports com.toy.anagrams.ui;
```

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

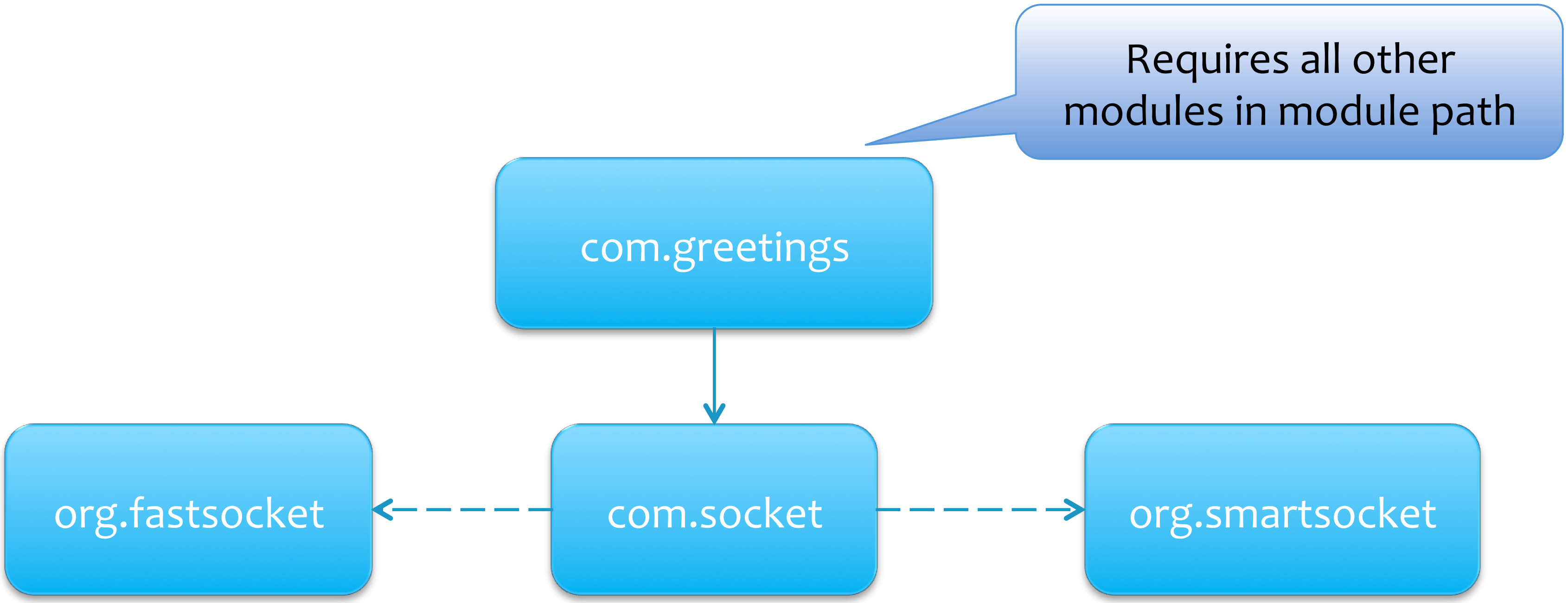
Loose-coupling





Loose-coupling

```
ServiceLoader<NetworkSocketProvider> sl =
    ServiceLoader.load(NetworkSocketProvider.class);
Iterator<NetworkSocketProvider> iter = sl.iterator();
```



Java 9 Modules and Services

```
module com.socket {
    exports com.socket;
    exports com.socket.spi;
    uses com.socket.spi.NetworkSocketProvider;
}
```

service

```
module org.fastsocket {
    requires com.socket;
    provides com.socket.spi.NetworkSocketProvider
             with org.fastsocket.FastNetworkSocketProvider;
}
```

service provider;
no packages are exported

```
module com.greetings {
    requires com.socket;
}
```

service consumer;
requires all service providers in module path

build/classes/META-INF/services/

Java 9 Modules & Services

* Java 6 uses a Query-based approach, the `ServiceLoader`:

```
ServiceLoader<Provider> serviceLoader =
    ServiceLoader.load(Provider.class);
for (Provider provider : serviceLoader) { return provider; }
```

```
ServiceLoader<Provider> serviceLoader =
    ServiceLoader.load(Provider.class).stream().filter(...);
```

* However, the `ServiceLoader` has a number of problems:
  * it isn't dynamic (you cannot install/uninstall a plugin/service at runtime)
  * it does all service loading at startup (as a result it requires longer startup time and more memory usage)
  * it cannot be configured; there is a standard constructor and it doesn't support factory methods
  * it doesn't allow for ranking/ordering, i.e. we cannot choose which service to load first

Java 9 Service Loader

* Java 9 modifications to Java 6 `ServiceLoader`:
  * No relative services; the new module-based service locator does not have relative behaviour
  * Ordering of services (as they were discovered) is lost
  * all service interfaces and implementations on the module path are flattened into a single, global namespace
  * No extensibility / customizability of service loading; the service layer provider must provide a fixed mapping of available services up front
  * multiple-site declarations; every module that uses a service must also declare that the service is being used in the module descriptor; no global layer-wide service registry

NetBeans 9 EA

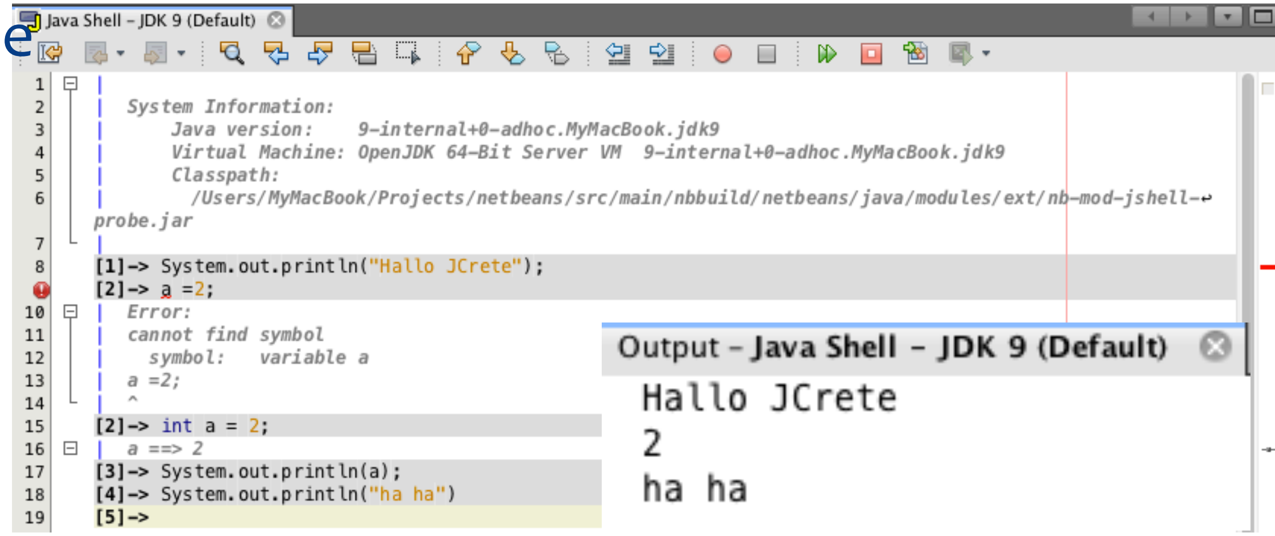
Getting started
May the source be with you

* Download JDK 9 Early Access **build** from https://jdk9.java.net/download/ page. Follow instructions how to build. Binary in `build/<os>-normal-server-release/jdk`.
* Download the latest NetBeans with JDK 9 support from http://wiki.netbeans.org/JDK9Support or build it from sources. Binary in `nbbuild/netbeans`.
* Configure it to run with JDK 8 or JDK 9 EA (`etc/netbeans.conf`).
* If you start NetBeans 9 with JDK 9 EA, `jshell` is enabled under **Tools** menu.
* Register the latest JDK 9 EA build as a *Java Platform* in NetBeans by means of **Tools →Java Platforms → Add Platform.**

JShell in NetBeans

- **Tools → Open Java Platform Shell**
- **Window → Output**

1. Semicolon is optional
2. NetBeans Shortcuts work! (e.g. `sout` → (tab))
3. Java expressions (e.g. `2+2`)
4. Forward reference
5. JShell API
6. `printf()`
7. `/help`

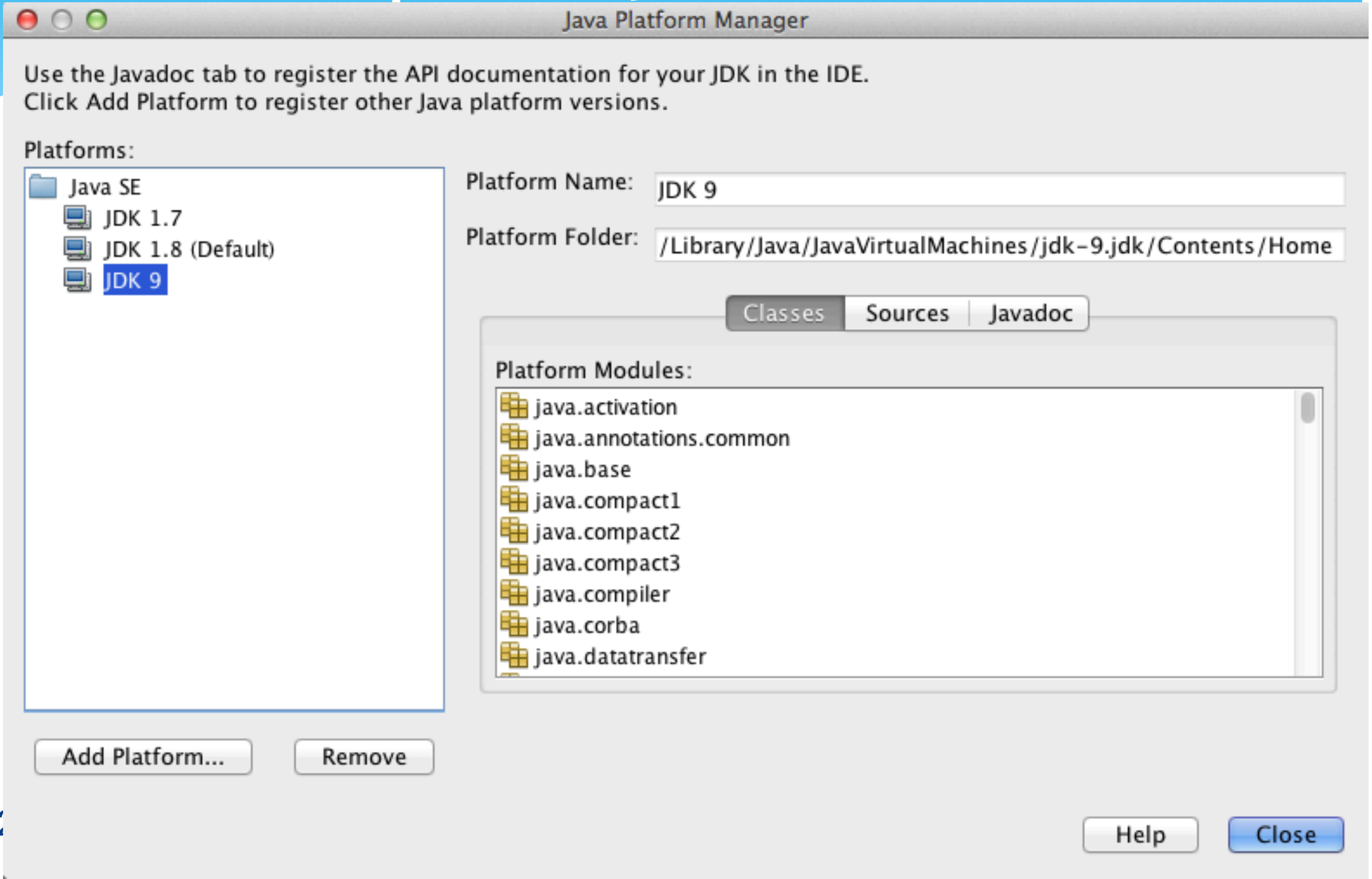


```

  System Information:
      Java version:    9-internal+0-adhoc.MyMacBook.jdk9
      Virtual Machine: OpenJDK 64-Bit Server VM  9-internal+0-adhoc.MyMacBook.jdk9
      Classpath:
        /Users/MyMacBook/Projects/netbeans/src/main/nbbuild/netbeans/java/modules/ext/nb-mod-jshell-↩
probe.jar

[1]-> System.out.println("Hallo JCrete");
[2]-> a =2;
  Error:
  cannot find symbol
    symbol:   variable a
  a =2;
  ^
[2]-> int a = 2;
  a ==> 2
[3]-> System.out.println(a);
[4]-> System.out.println("ha ha")
[5]->
```

Output – Java Shell – JDK 9 (Default)

```
Hallo JCrete
2
ha ha
```

Setup JDK 9 EA Platform



Java Platform Manager

Use the Javadoc tab to register the API documentation for your JDK in the IDE.
Click Add Platform to register other Java platform versions.

Platforms:

- Java SE
  - JDK 1.7
  - JDK 1.8 (Default)
  - JDK 9

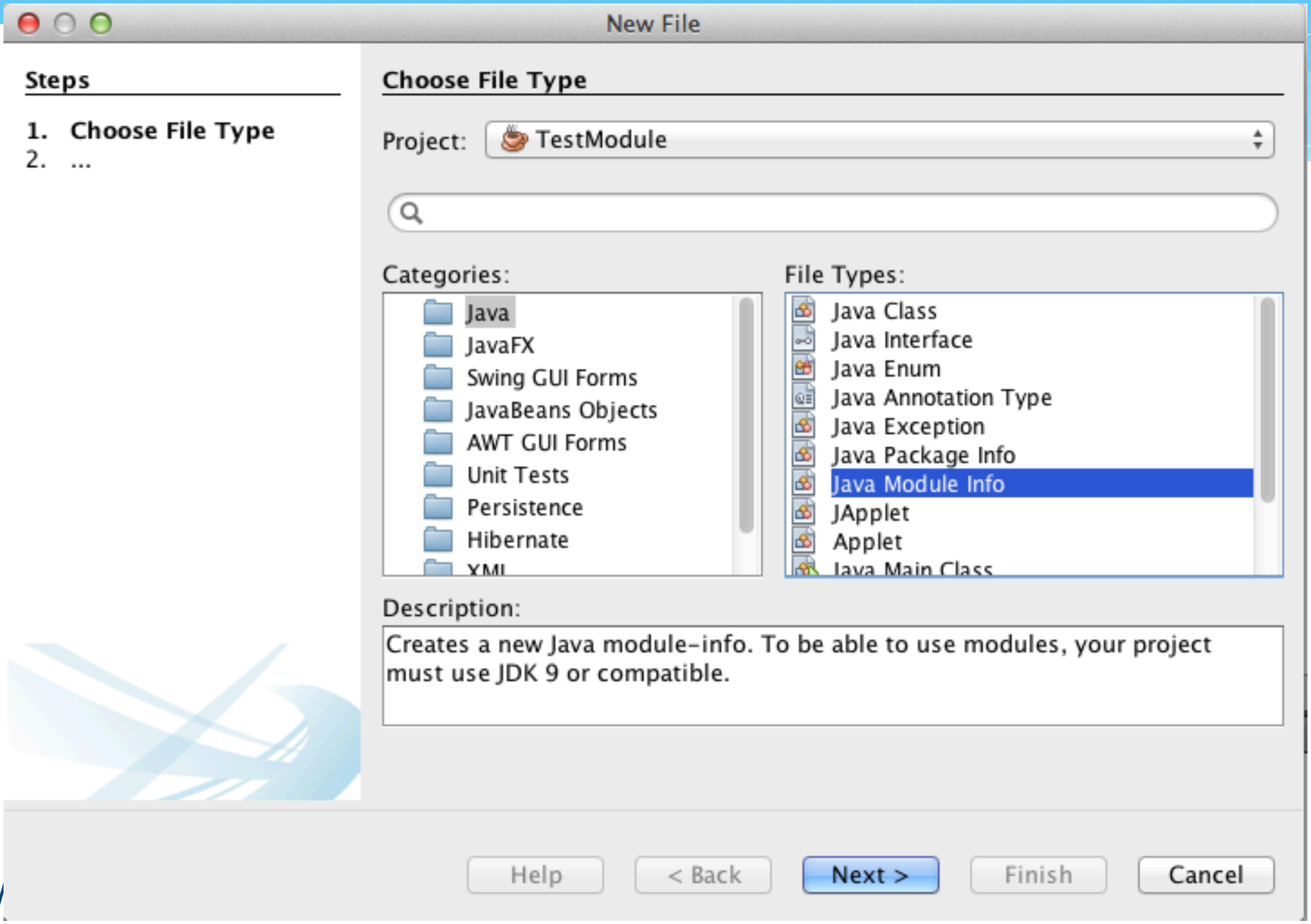
Platform Name: JDK 9

Platform Folder: /Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home

Platform Modules:

- java.activation
- java.annotations.common
- java.base
- java.compact1
- java.compact2
- java.compact3
- java.compiler
- java.corba
- java.datatransfer

Add module-info.java to a Java Project



Project: TestModule

Categories:
- Java
- JavaFX
- Swing GUI Forms
- JavaBeans Objects
- AWT GUI Forms
- Unit Tests
- Persistence
- Hibernate
- XML

File Types:
- Java Class
- Java Interface
- Java Enum
- Java Annotation Type
- Java Exception
- Java Package Info
- Java Module Info
- JApplet
- Applet
- Java Main Class

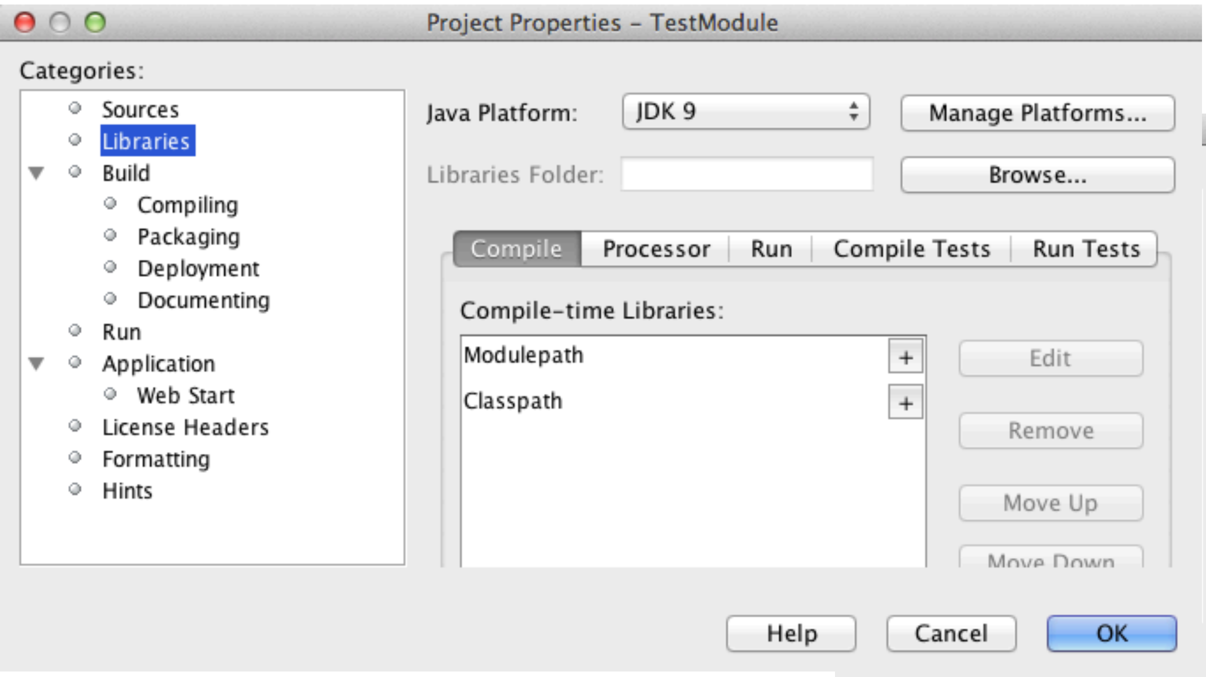
Description:

Creates a new Java module-info. To be able to use modules, your project must use JDK 9 or compatible.

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**



Project Properties – TestModule

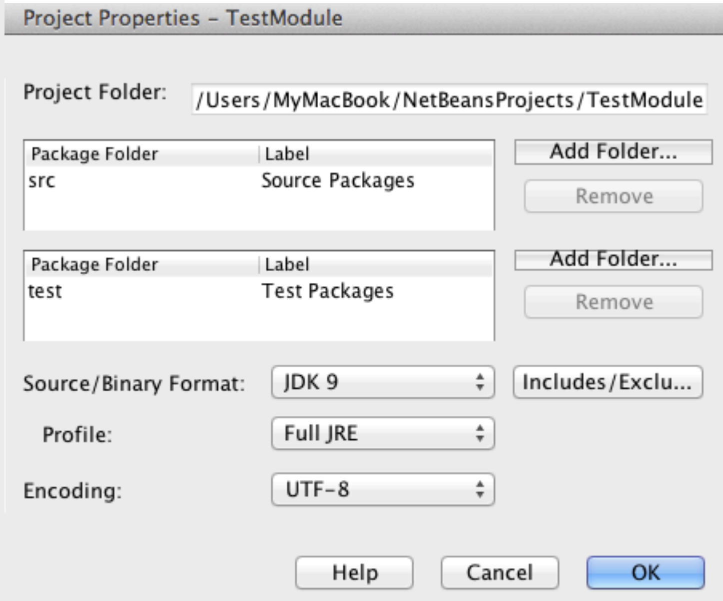
Categories:
- Sources
- Libraries
- Build
  - Compiling
  - Packaging
  - Deployment
  - Documenting
- Run
- Application
  - Web Start
- License Headers
- Formatting
- Hints

Java Platform: JDK 9 | Manage Platforms...

Libraries Folder: | Browse...

Compile | Processor | Run | Compile Tests | Run Tests

Compile-time Libraries:
- Modulepath
- Classpath



Project Properties – TestModule

Project Folder: /Users/MyMacBook/NetBeansProjects/TestModule

Package Folder	Label
src	Source Packages

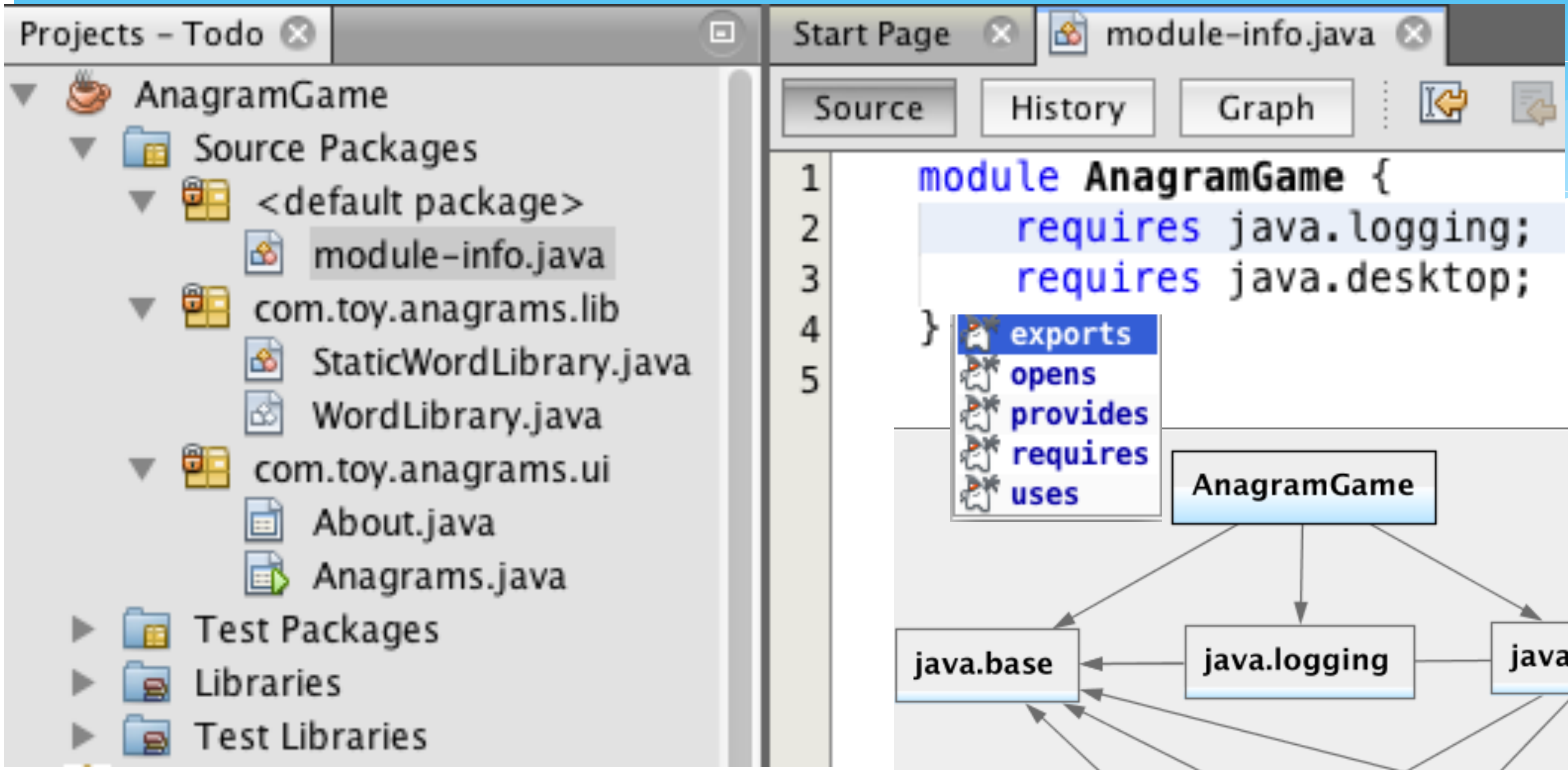
Package Folder	Label
test	Test Packages

Source/Binary Format: JDK 9

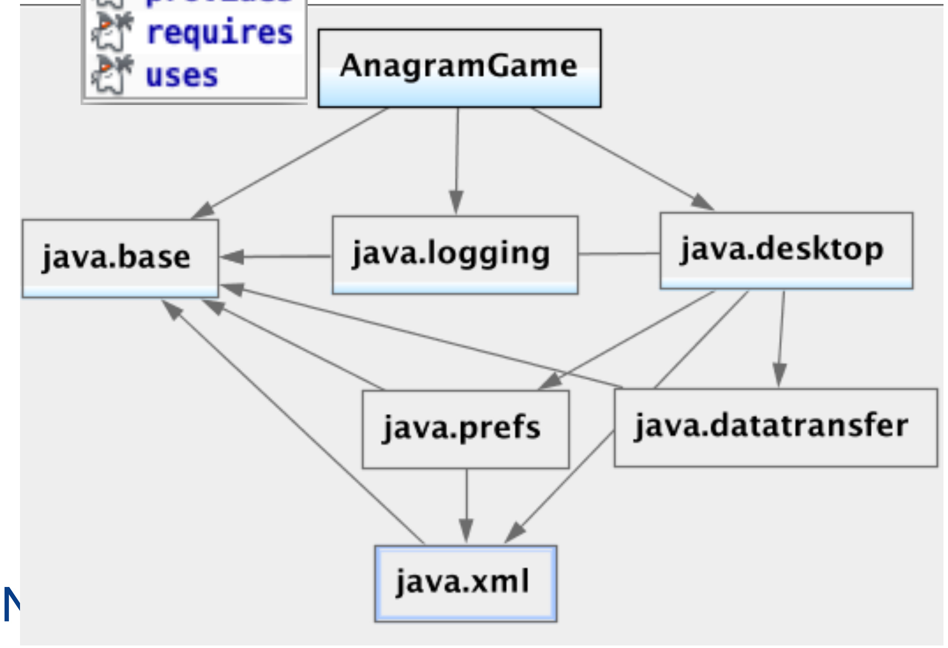
Profile: Full JRE

Encoding: UTF-8

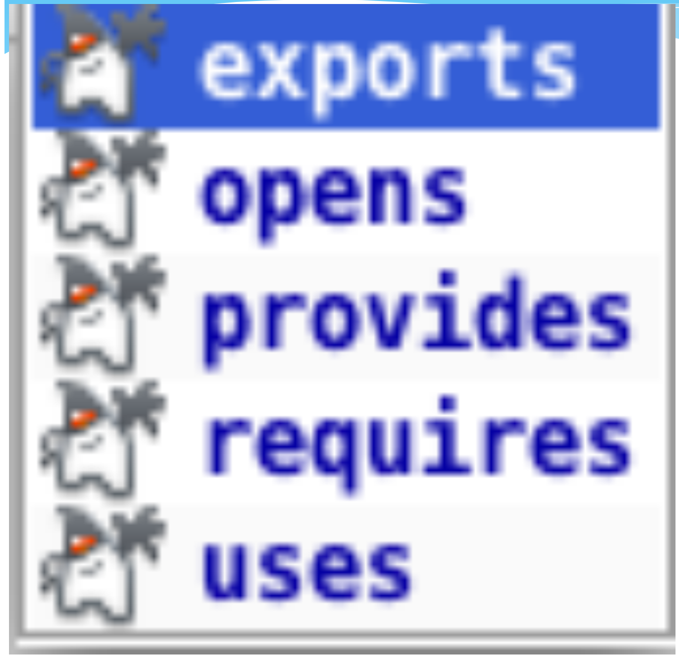
module-info.java



```
module AnagramGame {
    requires java.logging;
    requires java.desktop;
}
```

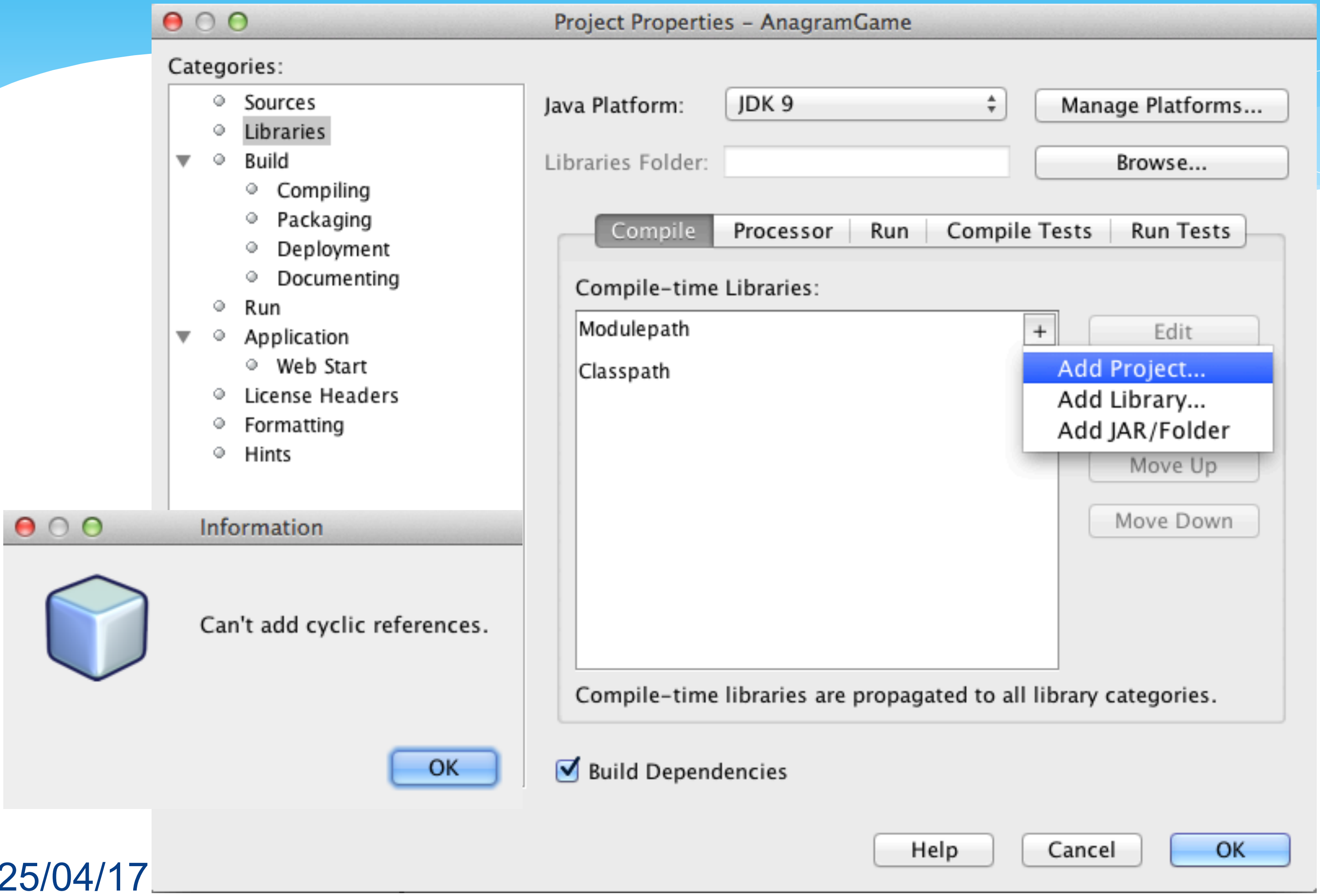


module-info.java


exports
opens
provides
requires
uses

* Exports a package
* Allows to use reflection on types in the package
* Provides a service provider
* Requires another module
* Uses a service

Module dependencies



Project Properties – AnagramGame

Categories:

- Sources
- Libraries
- Build
  - Compiling
  - Packaging
  - Deployment
  - Documenting
- Run
- Application
  - Web Start
- License Headers
- Formatting
- Hints

Java Platform: JDK 9

Libraries Folder:

Compile | Processor | Run | Compile Tests | Run Tests

Compile-time Libraries:

Modulepath

Classpath

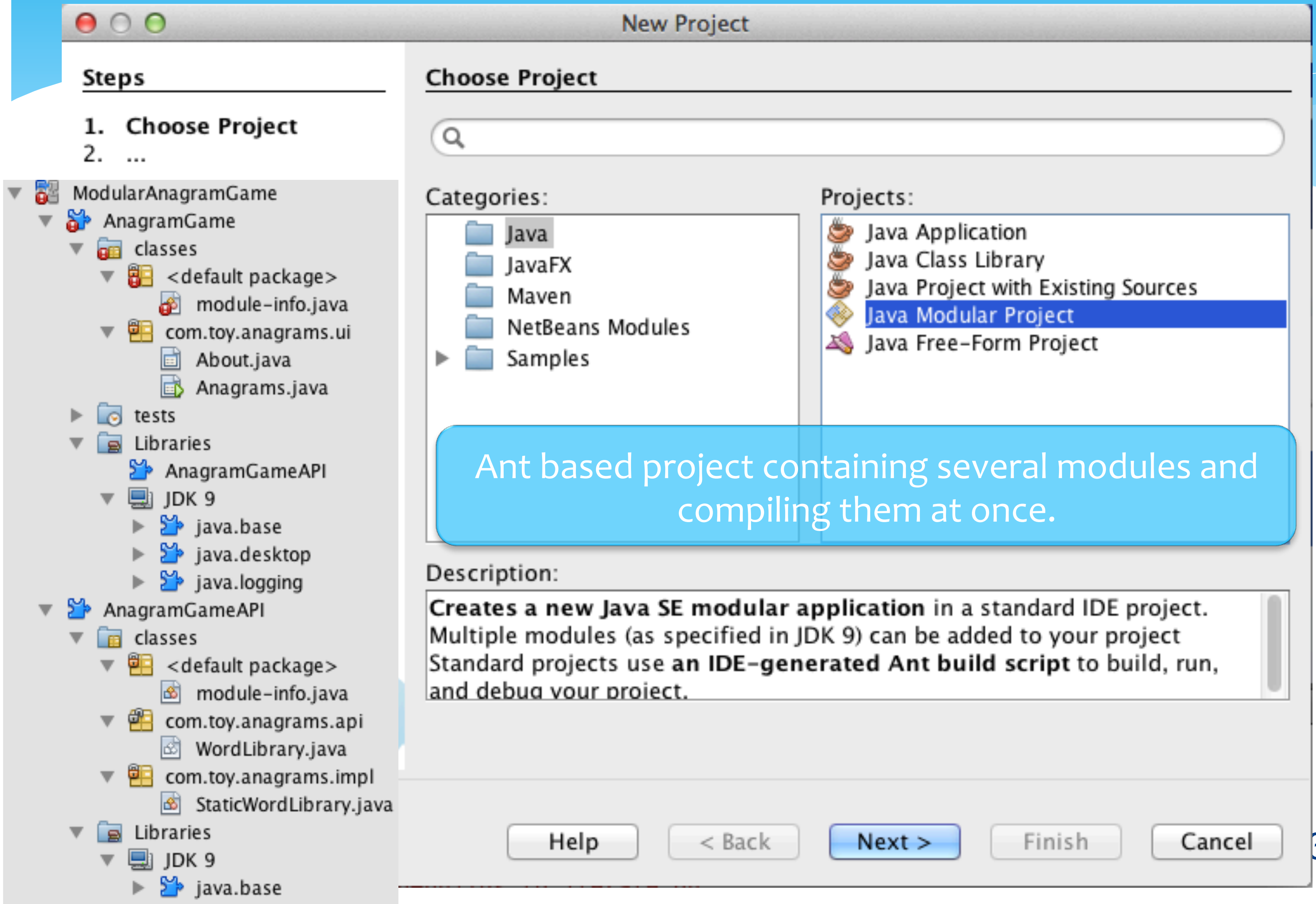
Compile-time libraries are propagated to all library categories.

[✓] Build Dependencies

Information

Can't add cyclic references.

Java SE Multi-Module Project



New Project

Steps

1. Choose Project
2. ...

Choose Project

Categories:

- Java
- JavaFX
- Maven
- NetBeans Modules
- Samples

Projects:

- Java Application
- Java Class Library
- Java Project with Existing Sources
- Java Modular Project
- Java Free-Form Project

Description:

Creates a new Java SE modular application in a standard IDE project. Multiple modules (as specified in JDK 9) can be added to your project Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

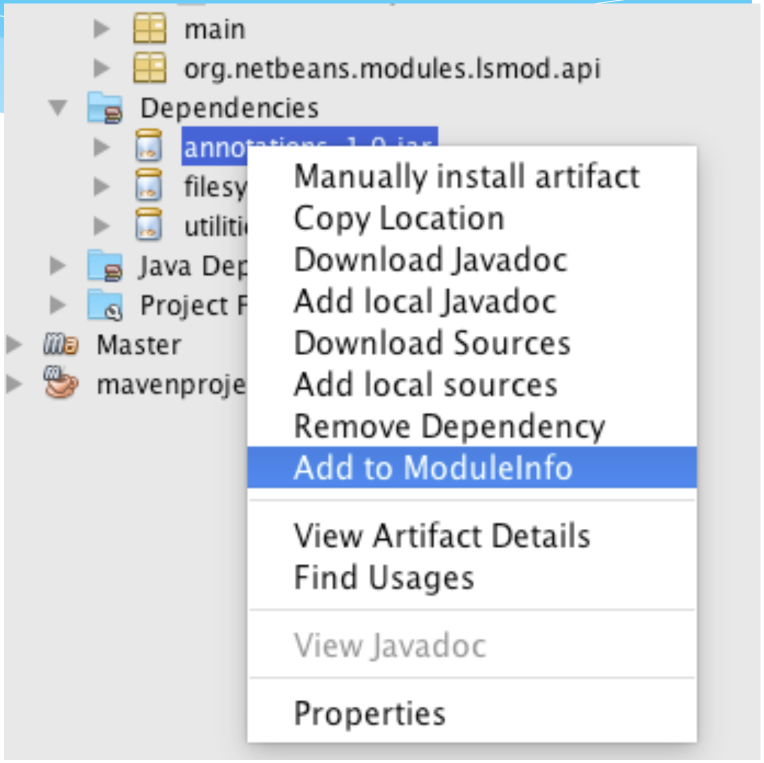
Help | < Back | Next > | Finish | Cancel

```
ModularAnagramGame
  AnagramGame
    classes
      <default package>
        module-info.java
      com.toy.anagrams.ui
        About.java
        Anagrams.java
    tests
    Libraries
      AnagramGameAPI
      JDK 9
        java.base
        java.desktop
        java.logging
  AnagramGameAPI
    classes
      <default package>
        module-info.java
      com.toy.anagrams.api
        WordLibrary.java
      com.toy.anagrams.impl
        StaticWordLibrary.java
    Libraries
      JDK 9
        java.base
```

Ant based project containing several modules and compiling them at once.

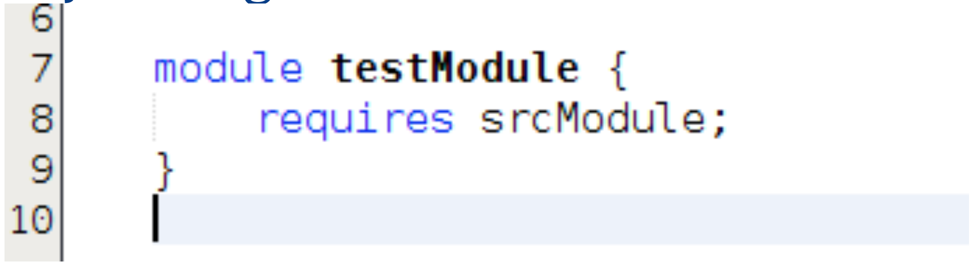
Maven support

- Maven projects do work ([Apache Maven Compiler Plugin 3.6.0](#))
- If `module-info.java` is present in Maven project then all Java libraries used in a project became JDK9 modules are placed into `MODULEPATH` by the Maven Compiler Plugin.
- It is also possible to add declared dependencies to `module_info.java` by right-clicking on **Dependencies.**



JUnit support

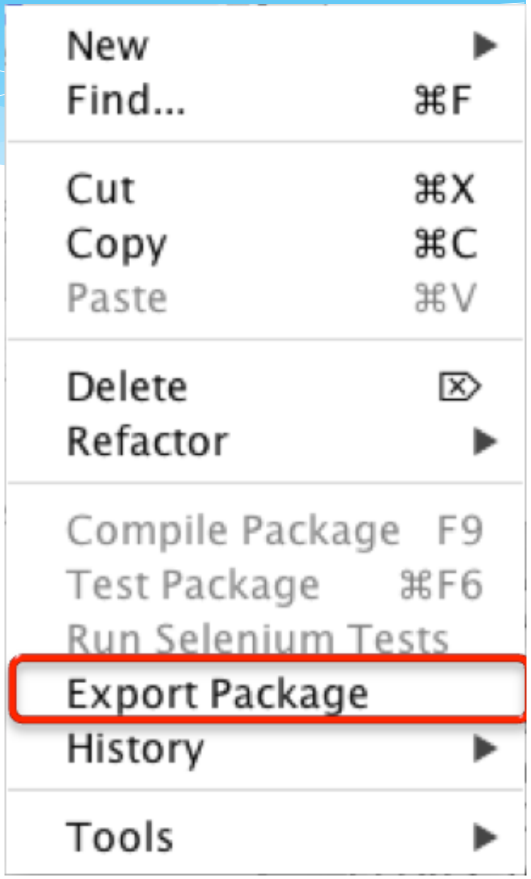
* NetBeans currently supports JUnit tests which are part of same module as tested sources.
* NetBeans also supports JUnit tests to be inside their own module project. This is the only case when two `module-info.java` files can be used in this type of project.
* Problems:
  * modules enforce module boundaries, only exported packages are seen from other modules.
  * module readability - dependency among modules.



```
module testModule {
    requires srcModule;
}
```

NetBeans 9 EA

- 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`)



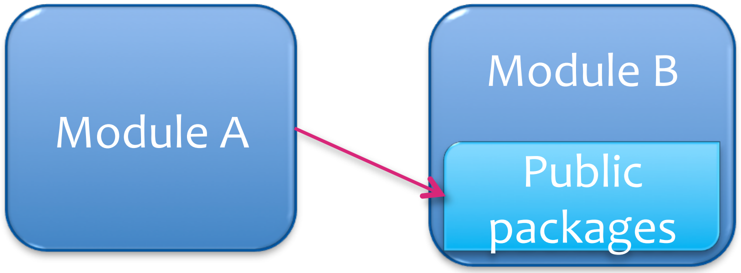
NetBeans Module API

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.

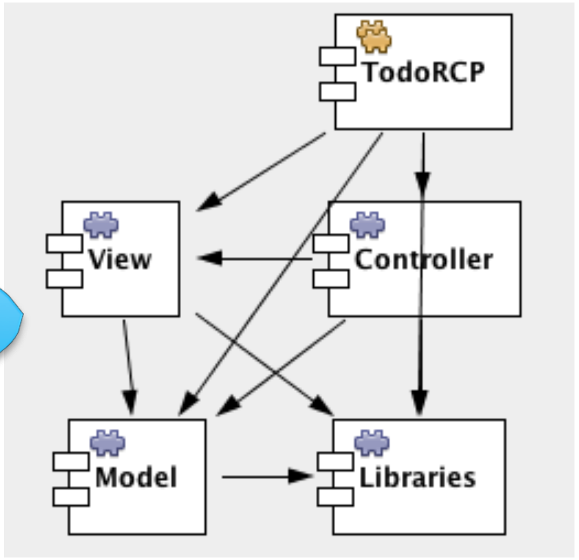


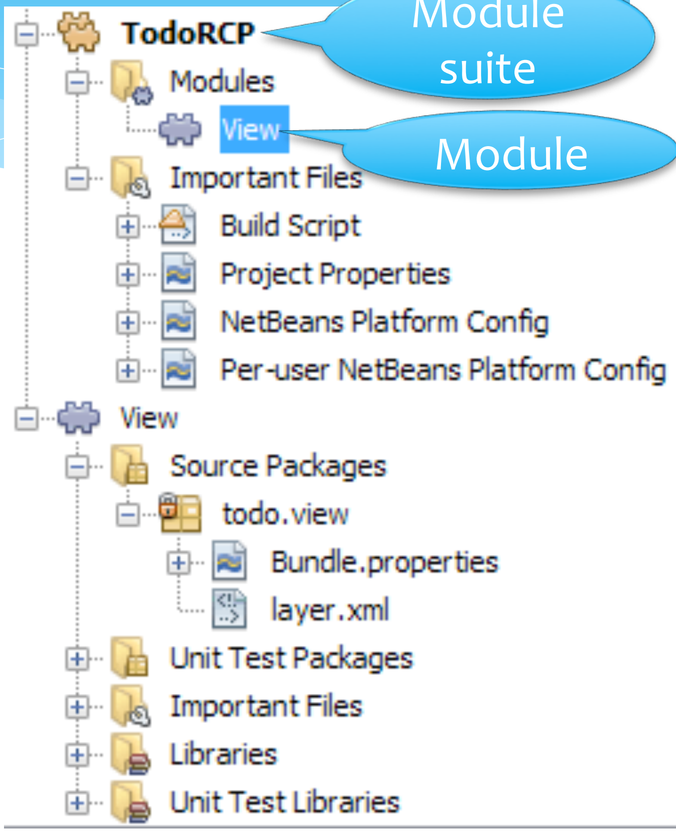
- In order to use or access code in another module:
  1. You must put Module B classes in a *public* package and assign a version number.
  2. Module A must declare a dependency on a specified version of Module B.

Modules and Module Suites

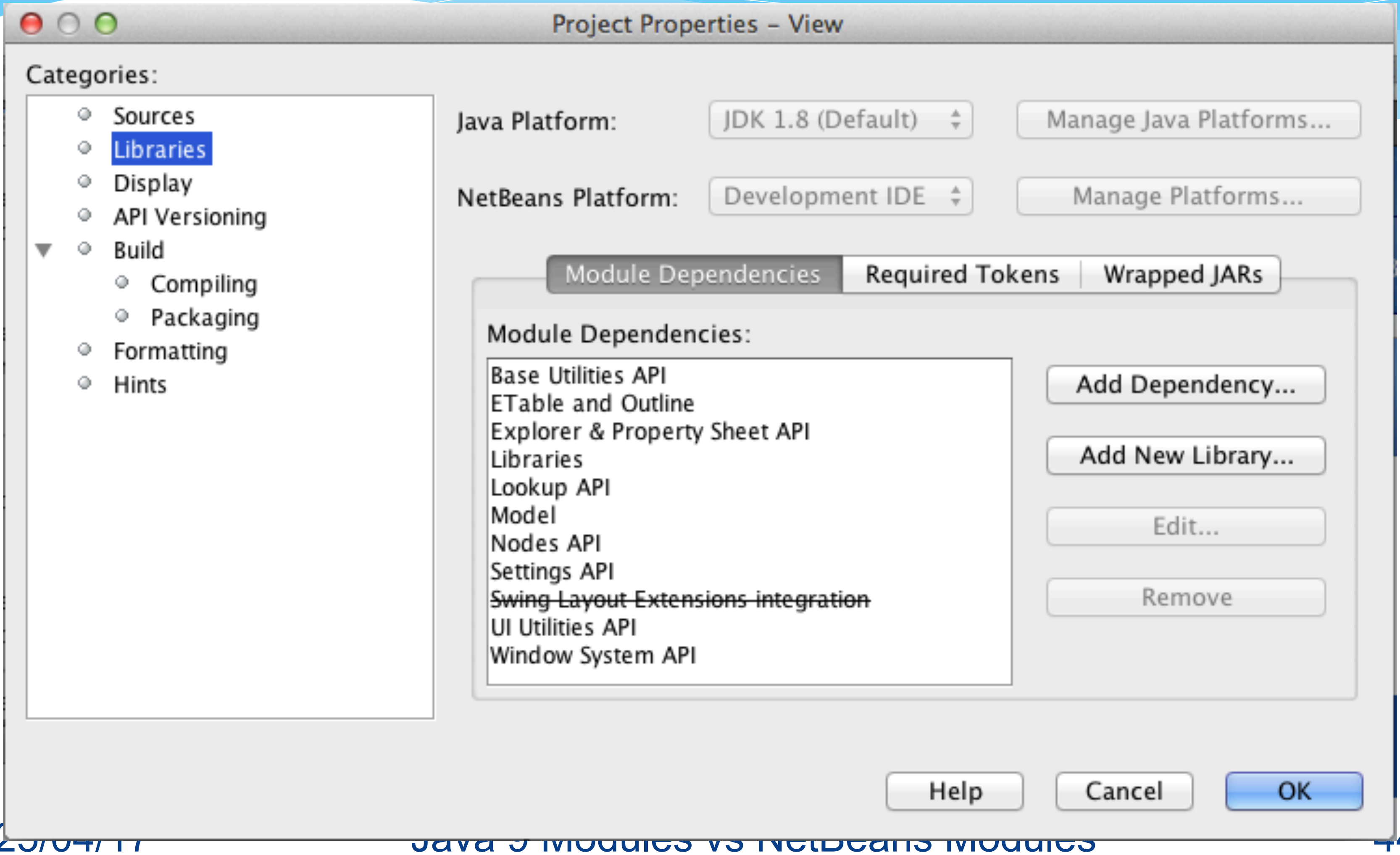
1. **File → New Project → NetBeans Modules → NetBeans Platform Application** creates a suite of modules
2. **Right-click on Modules → Add New**
   * View (todo.view)

Via DisplayDependencies plugin





Module Dependencies



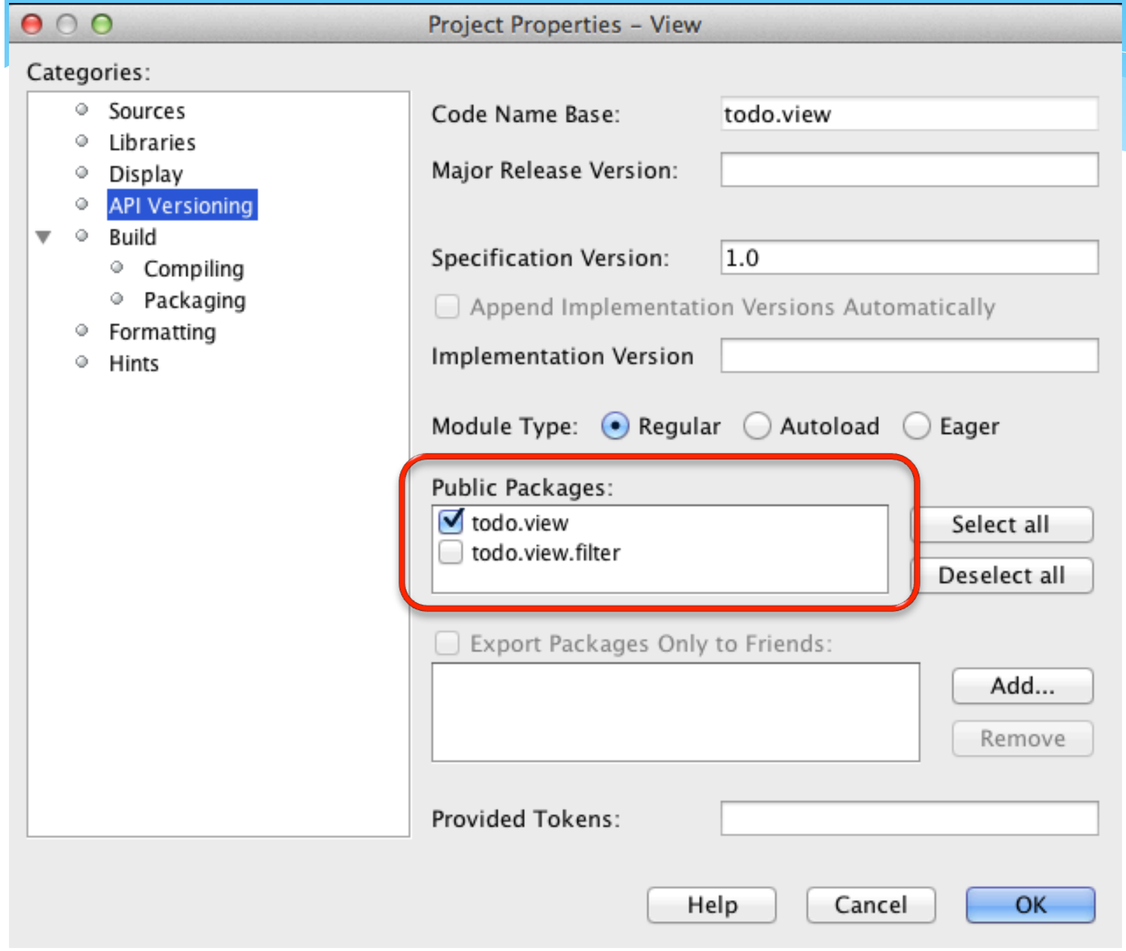
Java Platform: JDK 1.8 (Default)

NetBeans Platform: Development IDE

Module Dependencies:

Base Utilities API
ETable and Outline
Explorer & Property Sheet API
Libraries
Lookup API
Model
Nodes API
Settings API
~~Swing Layout Extensions integration~~
UI Utilities API
Window System API

Export packages



Project Properties – View

Categories:
- Sources
- Libraries
- Display
- API Versioning
- Build
  - Compiling
  - Packaging
- Formatting
- Hints

Code Name Base: todo.view

Major Release Version:

Specification Version: 1.0

☐ Append Implementation Versions Automatically

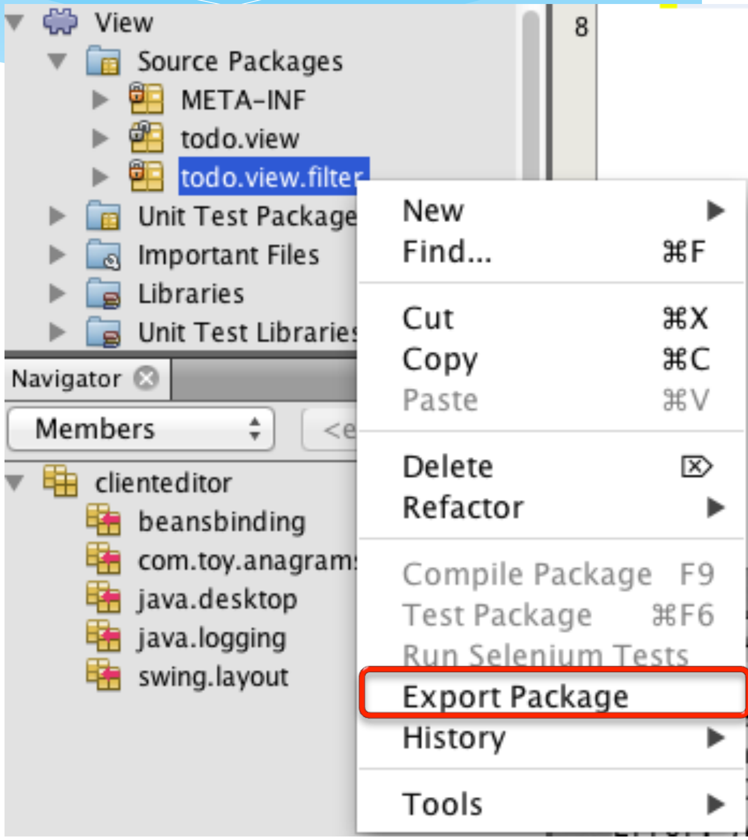
Implementation Version

Module Type: [✓ Regular] Autoload Eager

Public Packages:
- ☑ todo.view
- ☐ todo.view.filter

☐ Export Packages Only to Friends:

Provided Tokens:



NetBeans Modules & Services

* 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) { ... }
```

NetBeans Module & Lookup API

- You can use the NetBeans Module and the Lookup APIs outside of NetBeans to your own (non-NetBeans Platform) projects
- Simply copy the following jars that can be found inside `<NetBeans_Installation>/platform/lib/` to the `lib` folder of your project:
  - `org-openide-util-lookup.jar`
  - `org-openide-modules.jar`

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*	✗	✗
Services	`ServiceLoader`	`ServiceProvider`

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

- NetBeans 9 EA
- NetBeans 9 EA JDK 9 Support
- Ultimate Guide to Java 9, Sitepoint
- JDK 9 Feature Complete, JavaCodeGeeks
- Java 9 series: JShell, Voxxed
- Java 9 series: HTTP/2 Client, Voxxed
- Java 9 series: the JVM, Voxxed
- Java 9 series: HTML5 and Javadoc, Voxxed
- Java 9 series: Concurrency Updates, Voxxed
- Java 9 series: Variable Handles, Voxxed
- Java 9 series: Encapsulate Most Internal APIs, Voxxed
- Java 9 series: Multi-Release JAR Files, Voxxed
- Java 9 series: Segmented Code Cache, Voxxed
- Java 9 series: Convenience Factory Methods for Collections, Voxxed
- Critical Deficiencies in Jigsaw

References

* Bateman A. (2016), "Prepare for JDK 9", JavaOne.
* Bateman A. (2016), "Introduction to Modular Development", JavaOne.
* Bateman A. & Buckley A. (2016), "Advanced Modular Development", JavaOne.
* Buckley A. (2016), "Modules and Services", JavaOne.
* Buckley A. (2016), "Project Jigsaw: Under The Hood", JavaOne.
* Bateman A., Chung M., Reinhold M. (2016), "Project Jigsaw Hack Session", JavaOne.
* Evans, B. (2016), "An Early Look at Java 9 Modules", *Java Magazine*, Issue 26, January-February, pp.59-64.
* Mak S. & Bakker P. (2016), *Java 9 Modularity*, O'Reilly (Early Release)

References (cont.)

- Anderson G., Anderson P. (2014), *JavaFX Rich Client Programming on the NetBeans Platform*, Addison-Wesley.
- Boeck H. (2011), *The Definitive Guide to NetBeans Platform 7*, APress.
- Bourdeau T., Tulach J., Wielenga G. (2007), *Rich Client Programming*, Sun Microsystems.
- Epple A. (2009), "NetBeans Lookups Explained!", NetBeans DZone.
- Jenkov J. (2016), "ModRun Tutorial".
- Kostaras' blog, "Loose coupling".
- Wexbridge J. & Nyland W. (2014), *NetBeans Platform for Beginners*, LeanPub.

Questions

?