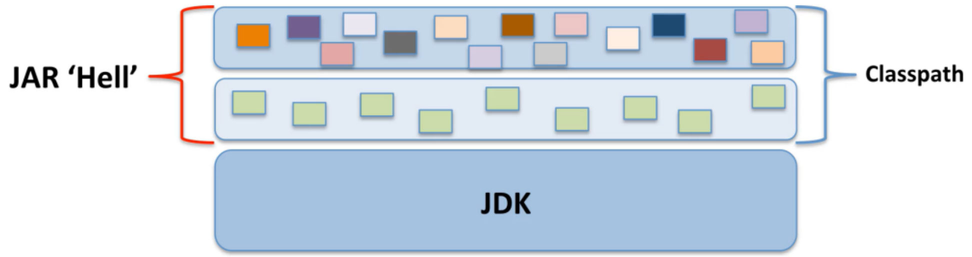
Java 11

Banuprakash C
Co-founder Lucida Technologies Pvt Ltd,
banuprakashc@yahoo.co.in
banu@lucidatechnologies.com

The Java Platform Module System

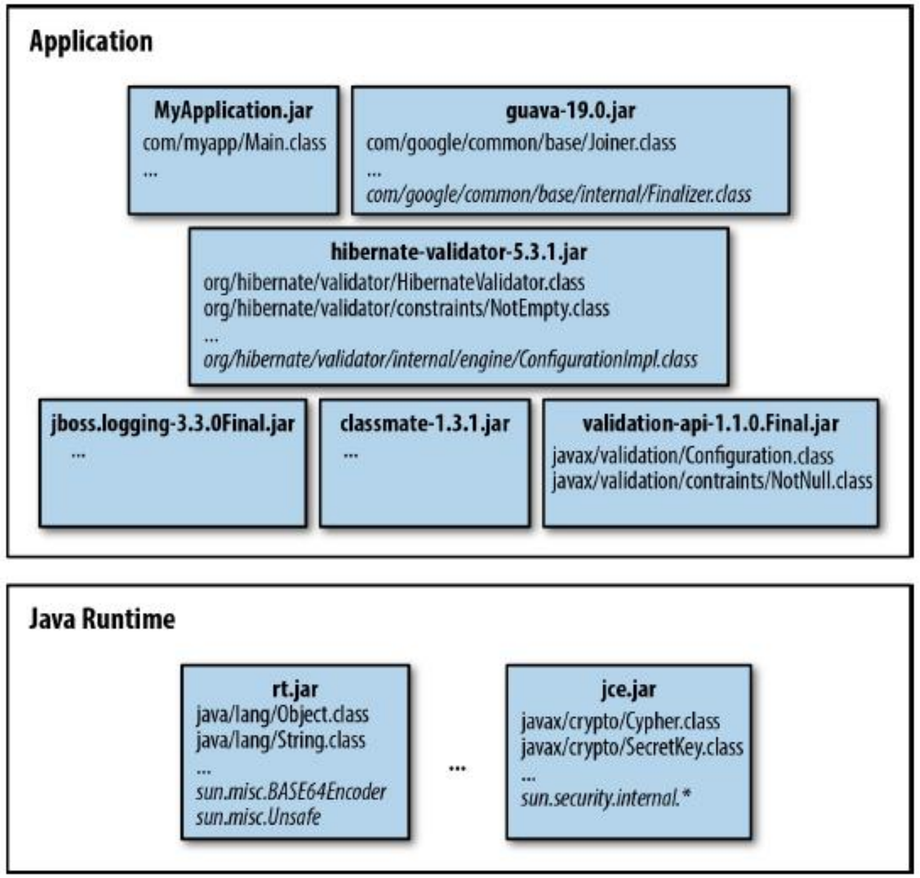
- Motivation
  - We have classes, interfaces, enums and annotations placed into packages
  - Converting packages into jars
  - Add 3rd party jars to make our code work
  - Place all these into classpath
  - Everything public in class libraries is accessible to everything else, even APIs
  - JARs can't define dependencies
  - Transitive dependencies
  - Shadowing and version conflicts
  - NoClassDefFoundError – at Runtime!!
  - JDK/JRE monolithic and very large, getting bigger with every new release



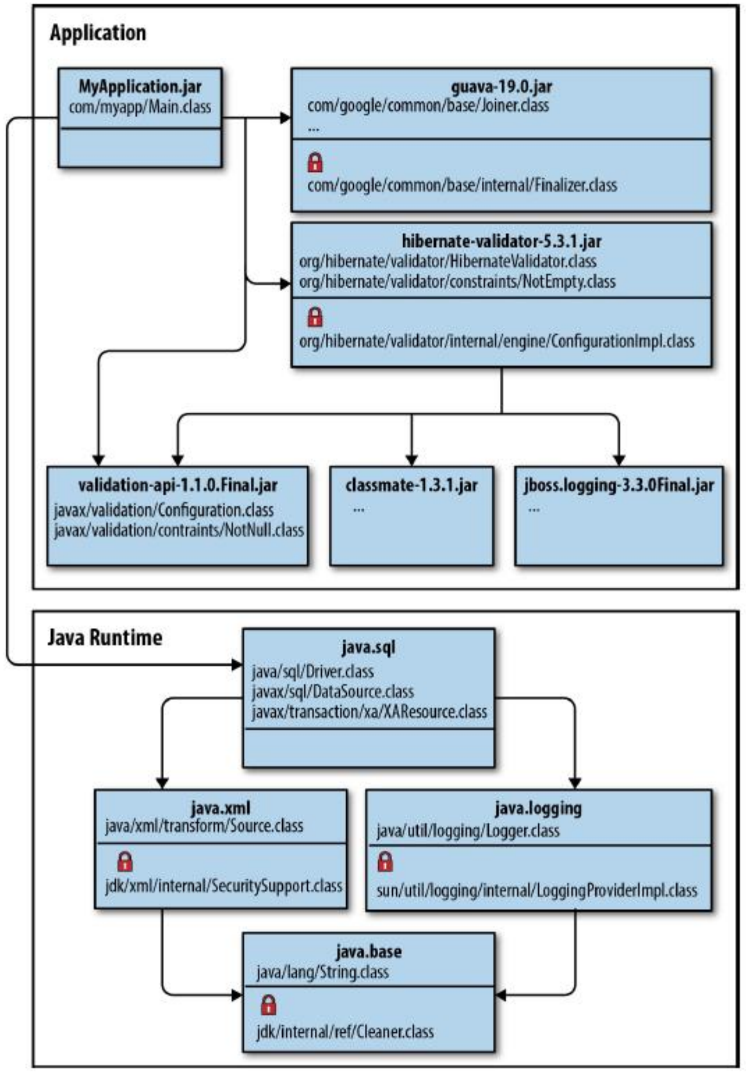
The Java Platform Module System

- Goals
  - Strong Encapsulation
    - Not everything public in classpath is available to everything else
  - Reliable Configuration
    - Configure dependencies so that we don’t have a situation where a resource is being called and we get NoClassDefError @ Runtime
  - Scalable Java Platform
    - 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.

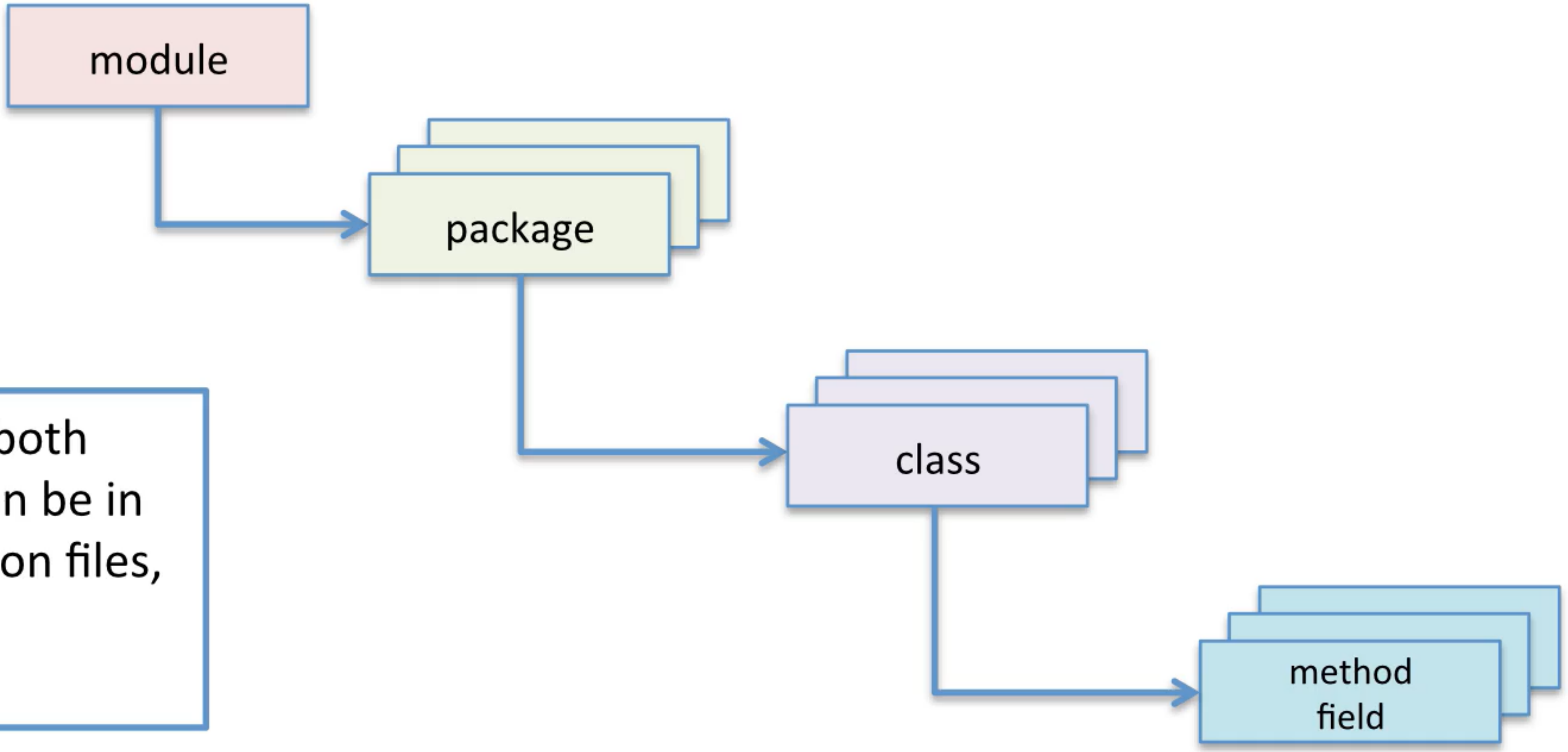
Java 8 and Java 9 Modular application


Application
MyApplication.jar
com/myapp/Main.class
...
guava-19.0.jar
com/google/common/base/Joiner.class
...
com/google/common/base/internal/Finalizer.class
hibernate-validator-5.3.1.jar
org/hibernate/validator/HibernateValidator.class
org/hibernate/validator/constraints/NotEmpty.class
...
org/hibernate/validator/internal/engine/ConfigurationImpl.class
jboss.logging-3.3.0Final.jar
...
classmate-1.3.1.jar
...
validation-api-1.1.0.Final.jar
javax/validation/Configuration.class
javax/validation/contraints/NotNull.class
Java Runtime
rt.jar
java/lang/Object.class
java/lang/String.class
...
sun.misc.BASE64Encoder
sun.misc.Unsafe
...
jce.jar
javax/crypto/Cypher.class
javax/crypto/SecretKey.class
...
sun.security.internal.*

Modular
Java 9

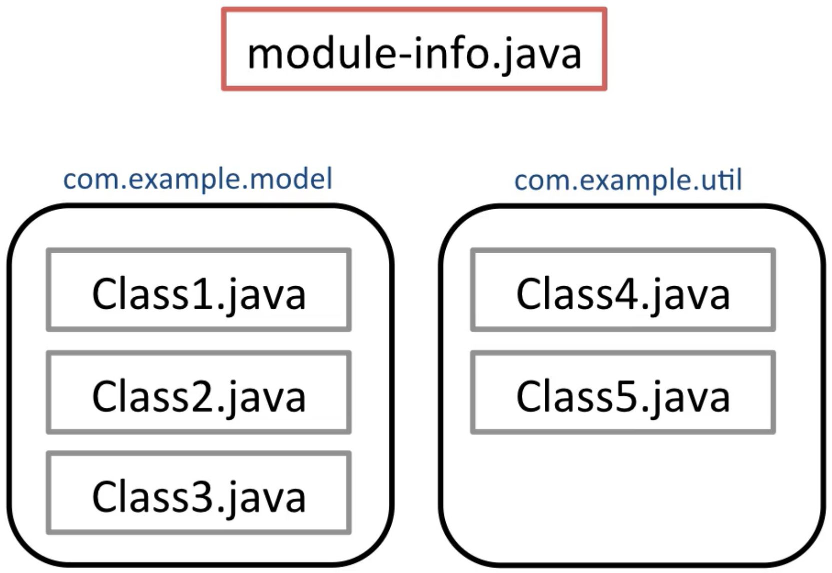

Application
MyApplication.jar
com/myapp/Main.class
guava-19.0.jar
com/google/common/base/Joiner.class
...
com/google/common/base/internal/Finalizer.class
hibernate-validator-5.3.1.jar
org/hibernate/validator/HibernateValidator.class
org/hibernate/validator/constraints/NotEmpty.class
org/hibernate/validator/internal/engine/ConfigurationImpl.class
validation-api-1.1.0.Final.jar
javax/validation/Configuration.class
javax/validation/contraints/NotNull.class
classmate-1.3.1.jar
...
jboss.logging-3.3.0Final.jar
...
Java Runtime
java.sql
java/sql/Driver.class
javax/sql/DataSource.class
javax/transaction/xa/XAResource.class
java.xml
java/xml/transform/Source.class
jdk/xml/internal/SecuritySupport.class
java.logging
java/util/logging/Logger.class
sun/util/logging/internal/LoggingProviderImpl.class
java.base
java/lang/String.class
jdk/internal/ref/Cleaner.class

Java 9 modules



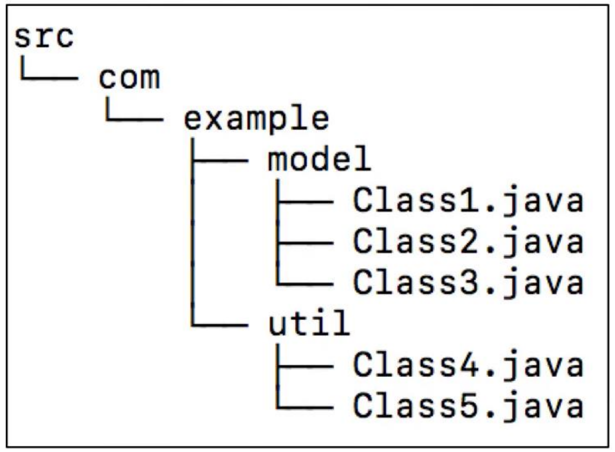
A module can contain both code and data. Data can be in the form of configuration files, native code, and other resources.

How to declare a Module?



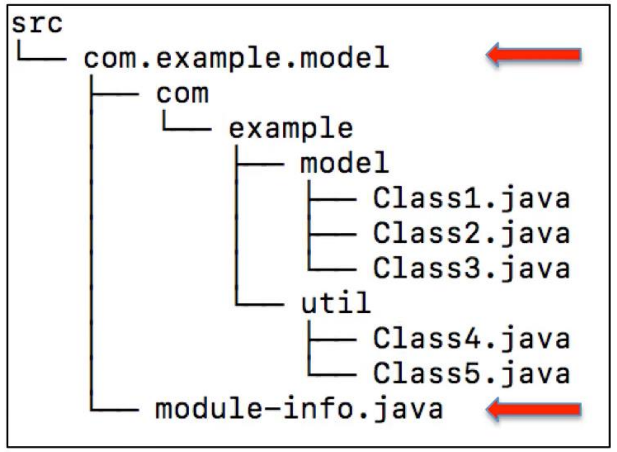
Directory Structure

Without Modules



```
src
└── com
    └── example
        ├── model
        │   ├── Class1.java
        │   ├── Class2.java
        │   └── Class3.java
        └── util
            ├── Class4.java
            └── Class5.java
```

With Modules



```
src
└── com.example.model
    ├── com
    │   └── example
    │       ├── model
    │       │   ├── Class1.java
    │       │   ├── Class2.java
    │       │   └── Class3.java
    │       └── util
    │           ├── Class4.java
    │           └── Class5.java
    └── module-info.java
```

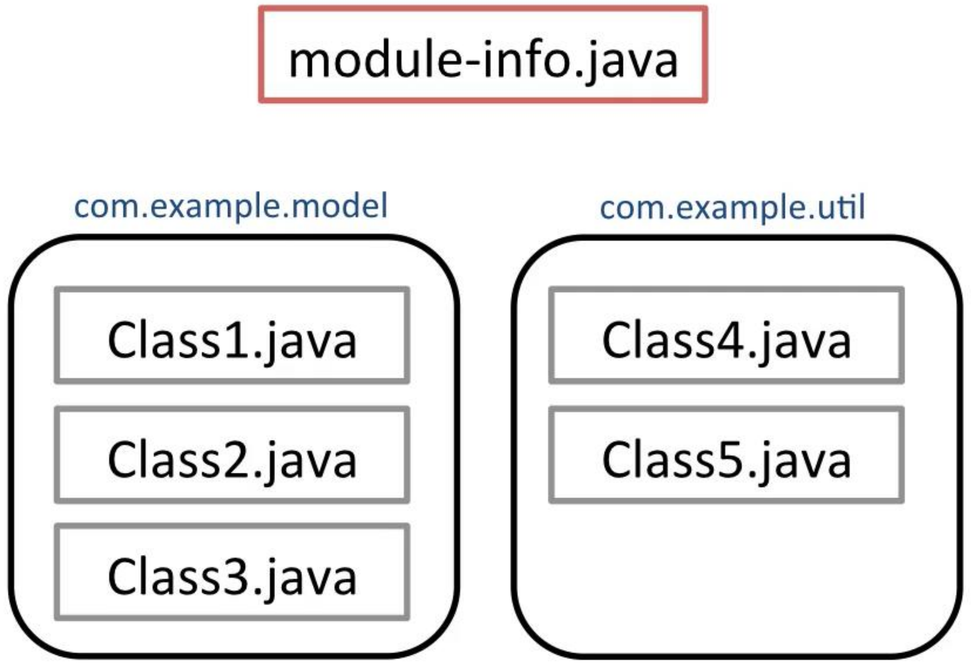
Modules

Module Directives

```
module com.example.model {
```

- What other **modules** do I **require**?
- What **packages** will I **export** to other **modules**?
- What services do I **provide** to to other **modules**?
- What services do I **use** from other **modules** ?
- Will my **packages** be **open** to reflection?

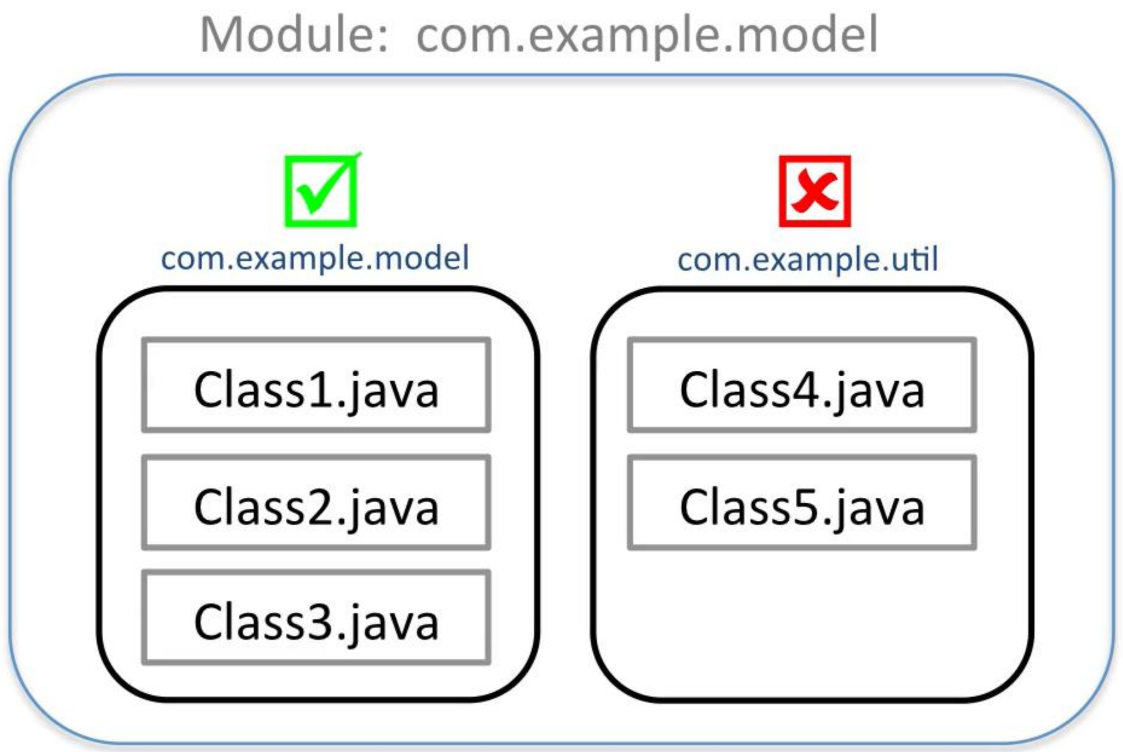
```
}
```



Module Directives

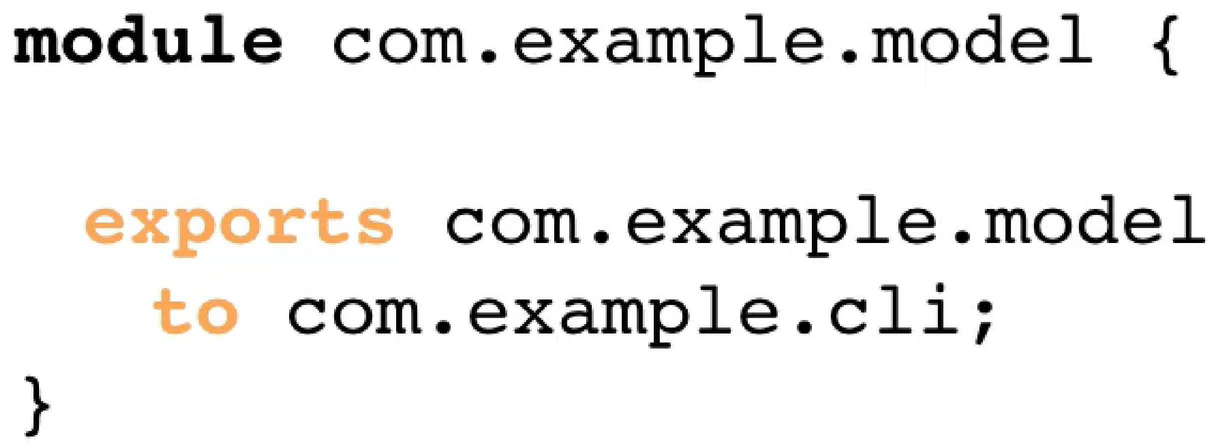
requires and exports

```
module com.example.cli {
  requires com.example.model;

}


module com.example.model {

  exports com.example.model;
}
```



Module Directives

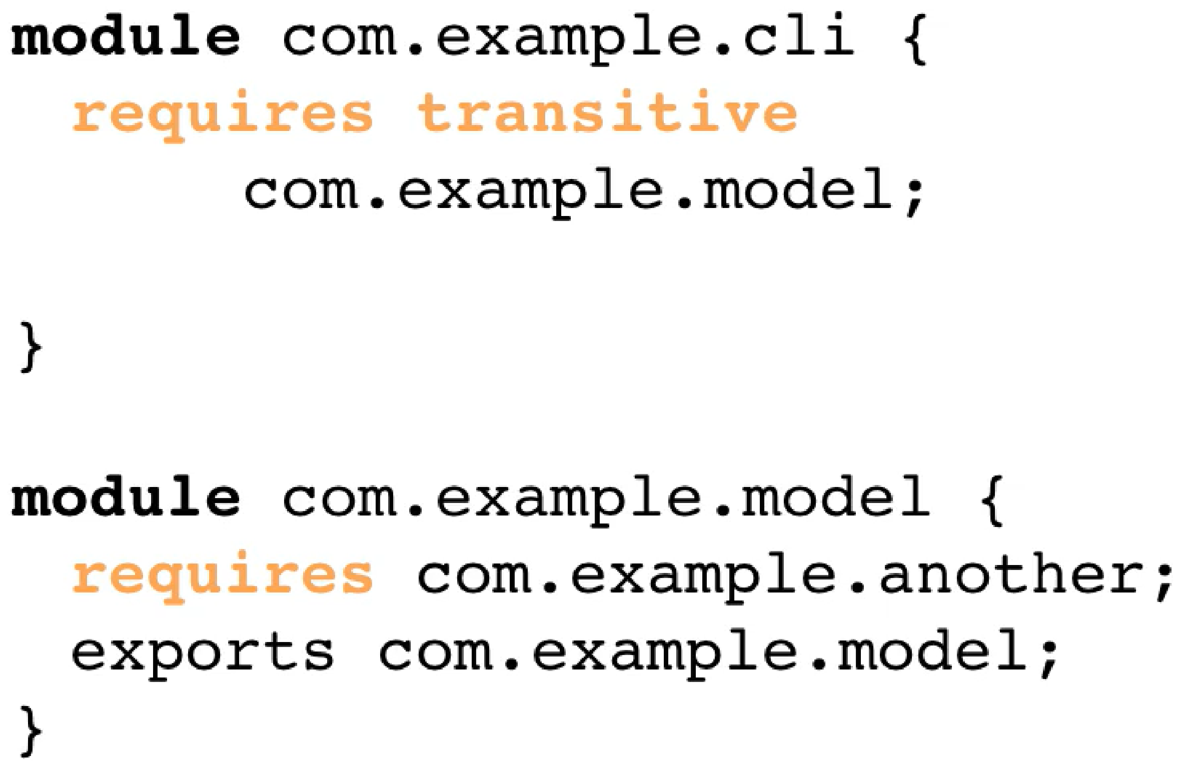
- exports to
  - Specify that only a certain module can access the given module



```
module com.example.model {

  exports com.example.model
    to com.example.cli;
}
```

Module Directives

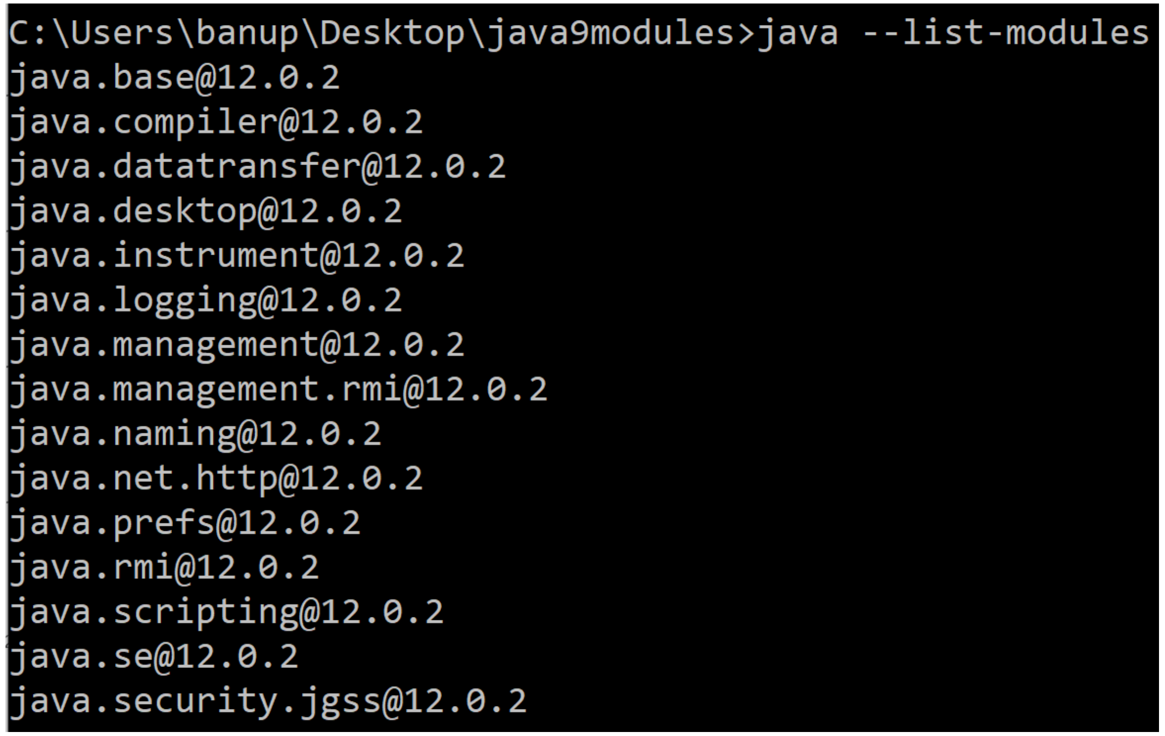
- requires transitive
  - This means that any module that reads your module implicitly also reads the *transitive* module



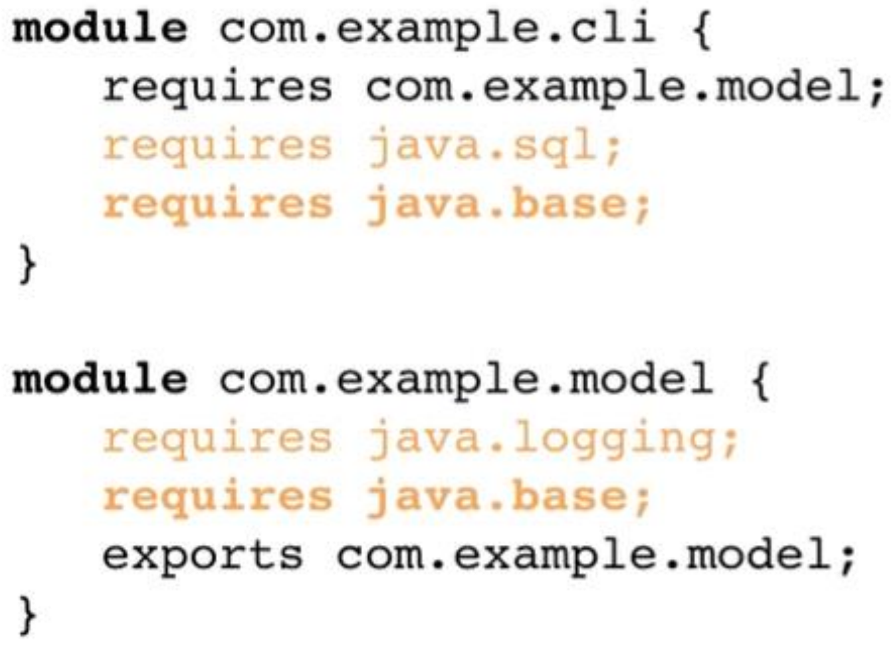
```
module com.example.cli {
  requires transitive
        com.example.model;

}

module com.example.model {
  requires com.example.another;
  exports com.example.model;
}
```

Module Directives

- JDK Modules

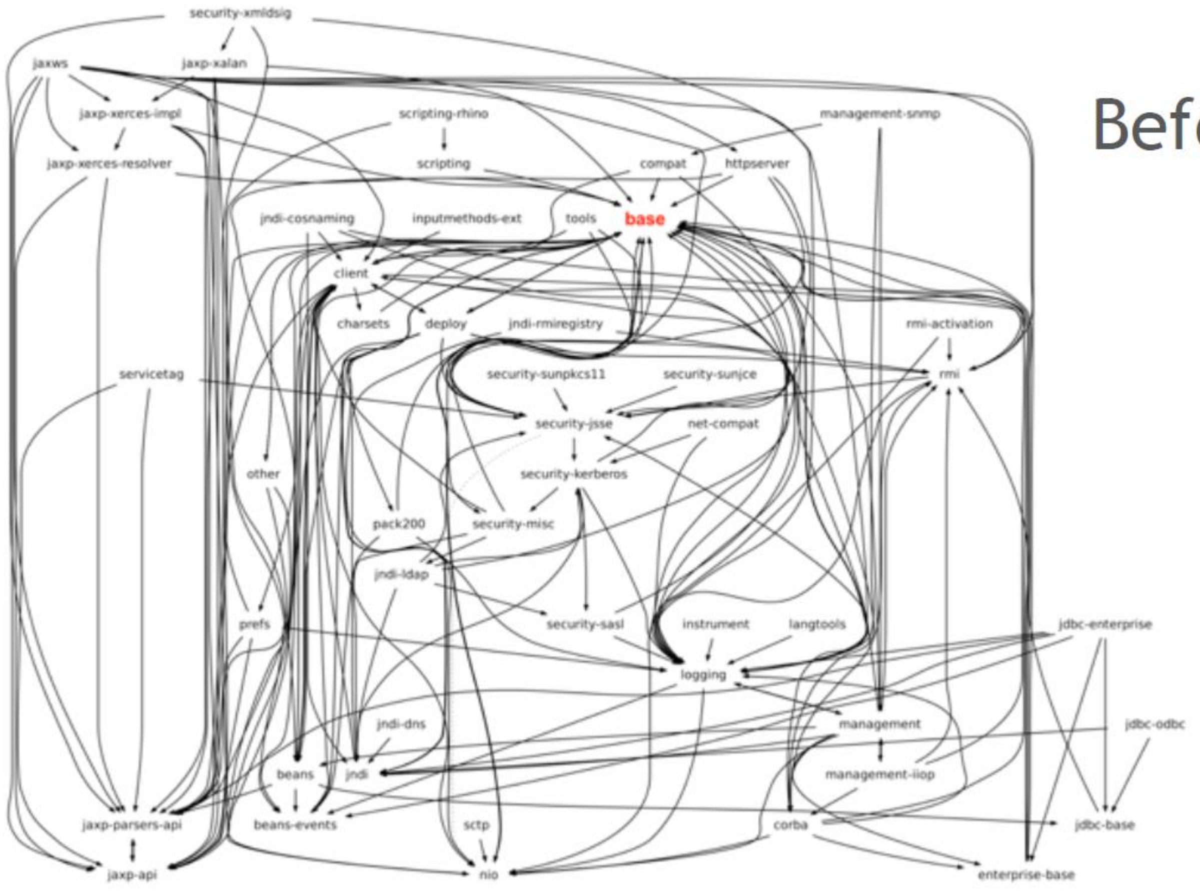


```
C:\Users\banup\Desktop\java9modules>java --list-modules
java.base@12.0.2
java.compiler@12.0.2
java.datatransfer@12.0.2
java.desktop@12.0.2
java.instrument@12.0.2
java.logging@12.0.2
java.management@12.0.2
java.management.rmi@12.0.2
java.naming@12.0.2
java.net.http@12.0.2
java.prefs@12.0.2
java.rmi@12.0.2
java.scripting@12.0.2
java.se@12.0.2
java.security.jgss@12.0.2
```



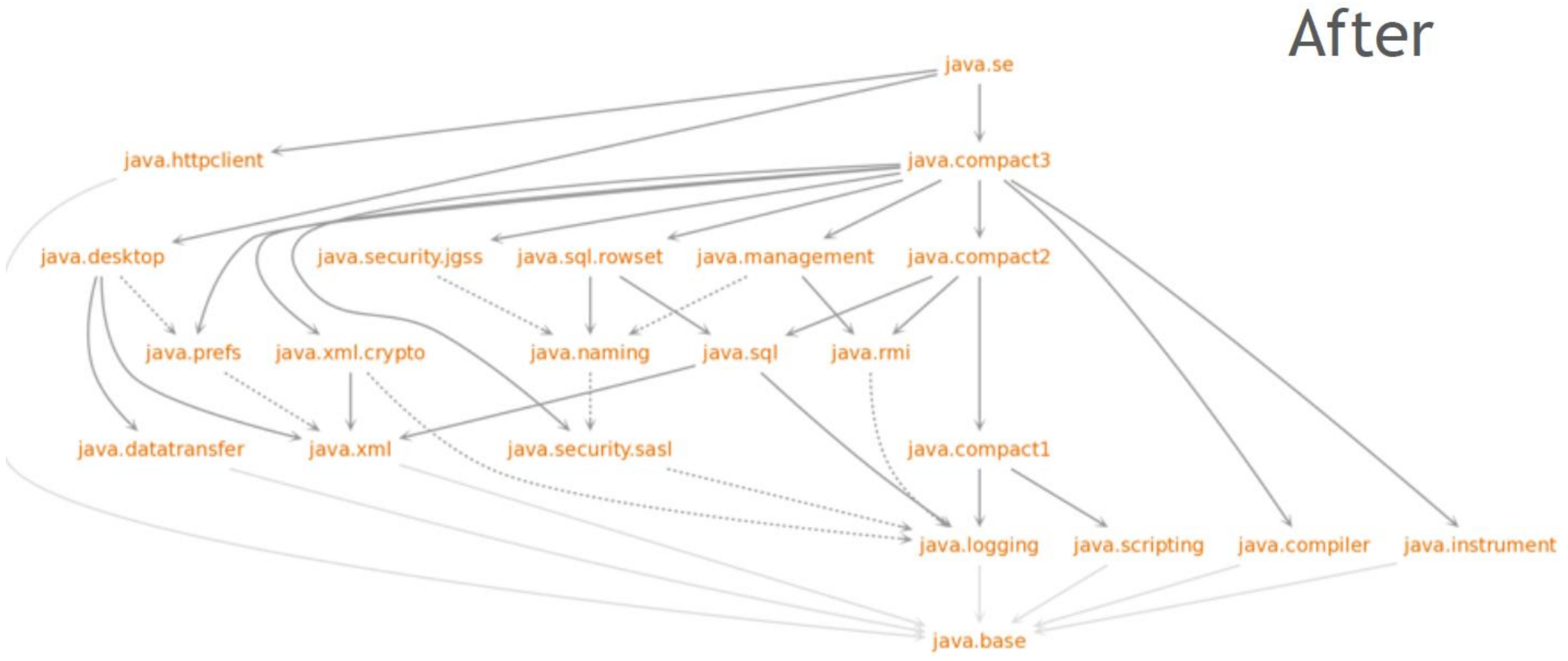
```
module com.example.cli {
    requires com.example.model;
    requires java.sql;
    requires java.base;
}

module com.example.model {
    requires java.logging;
    requires java.base;
    exports com.example.model;
}
```

Java 8 Jars

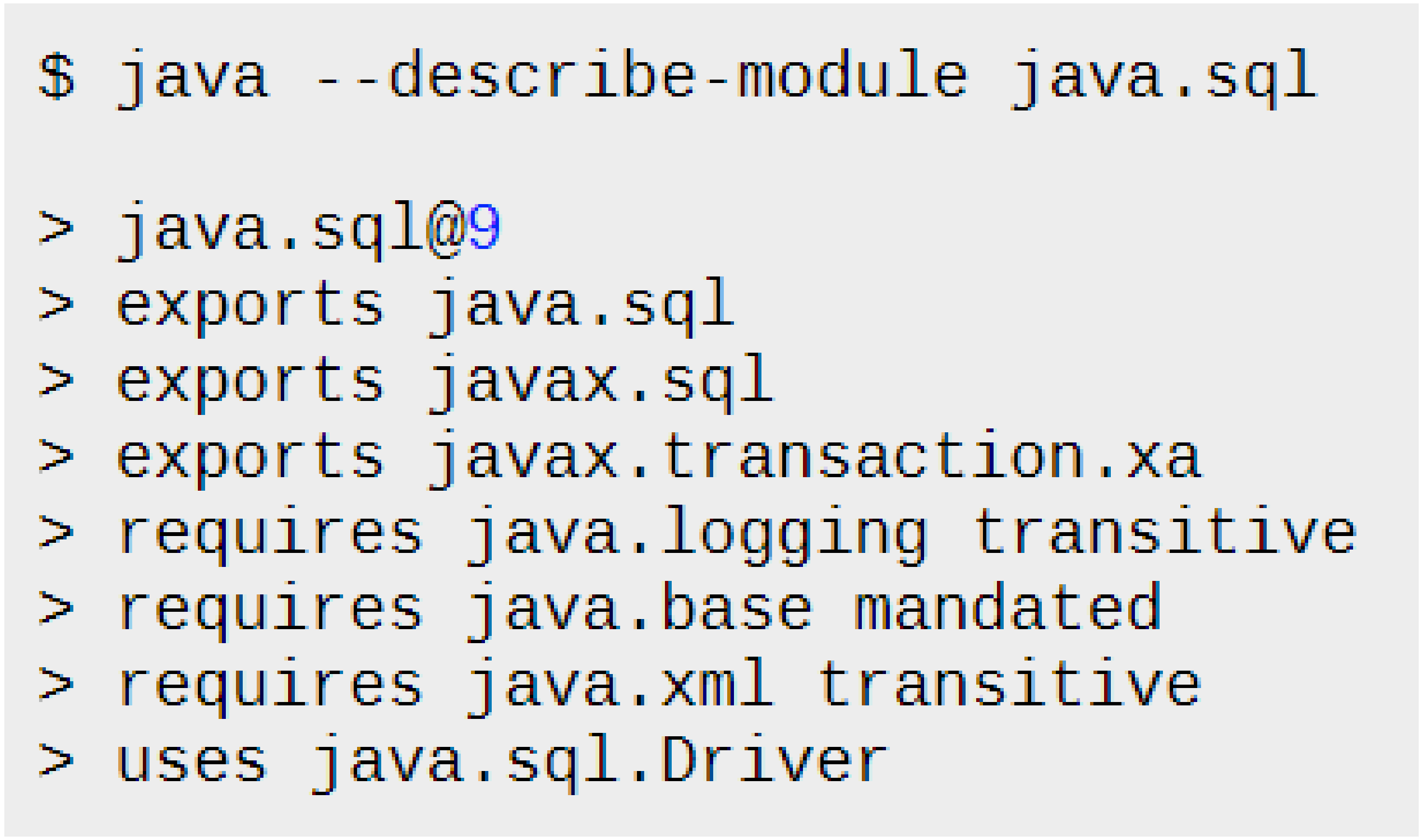


Before

Java 9 Modules



Java 9 Modules



```
$ java --describe-module java.sql

> java.sql@9
> exports java.sql
> exports javax.sql
> exports javax.transaction.xa
> requires java.logging transitive
> requires java.base mandated
> requires java.xml transitive
> uses java.sql.Driver
```

Commands

- **Compile**
  - **Windows:**
    - **dir /S /B src*.java > sources.txt**
    - **javac --module-source-path src -d out @sources.txt**
  - **Linux/Mac**
    - **javac   --module-source-path src -d out $(find src -name *.java)**
- **Run**
  - **java --module-path out --module modulename/package.classname**
- **Jars**
  - **jar --create --file=jars/nameofjar.jar -C out/modulelocation .**
- **Describe module**
  - **jar --describe-module --file= nameofjar.jar**
- **Jlink**
  - **jlink --module-path C:\jdk-12.0.2\jmods;./out --add-modules client.module,com.adobe.employee --output myappimage --launcher MYAPP=client.module/client.Main**
  - **Execute:**
    - **myappimage\bin>java –list-modules**
    - **myappimage\bin>myapp**

Java 9 Modules

- The module system leans heavily on a module’s name. Conflicting or evolving names in particular cause trouble, so it is important that the name is:
  - globally unique
  - stable
- The best way to achieve that is the reverse-domain naming scheme that is already commonly used for packages:

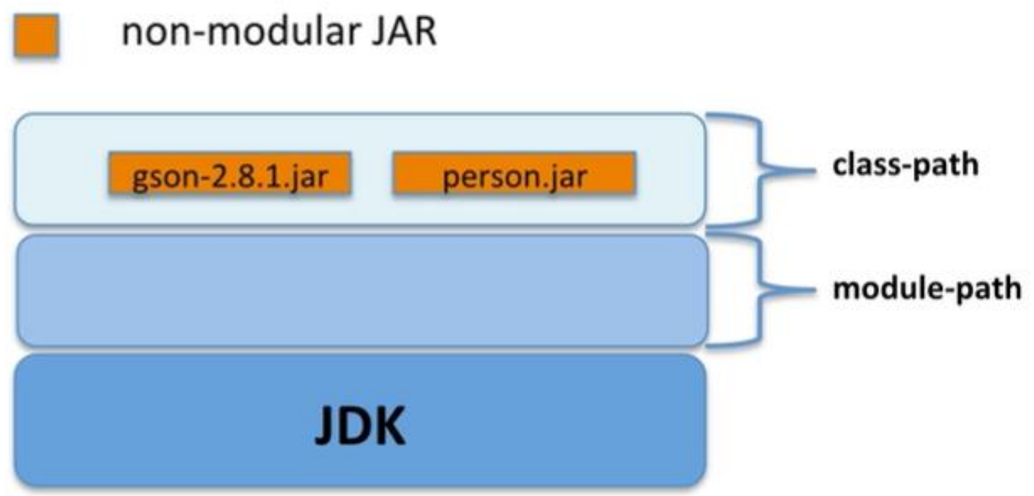
```
module com.example.demo.customermodule {


}
```

Java 9 Modules

- The *module path* is a list whose elements are artifacts or directories that contain artifacts.
- Depending on the operating system, module path elements are either separated by : (Unix-based) or ; (Windows).
- It is used by the module system to locate required modules that are not found among the platform modules.
- Both javac and java as well as other module-related commands can process it
    - the command line options are --**module**-path and -p.
- All artifacts on the module path are turned into modules. This is even true for plain JARs, which get turned into automatic modules.

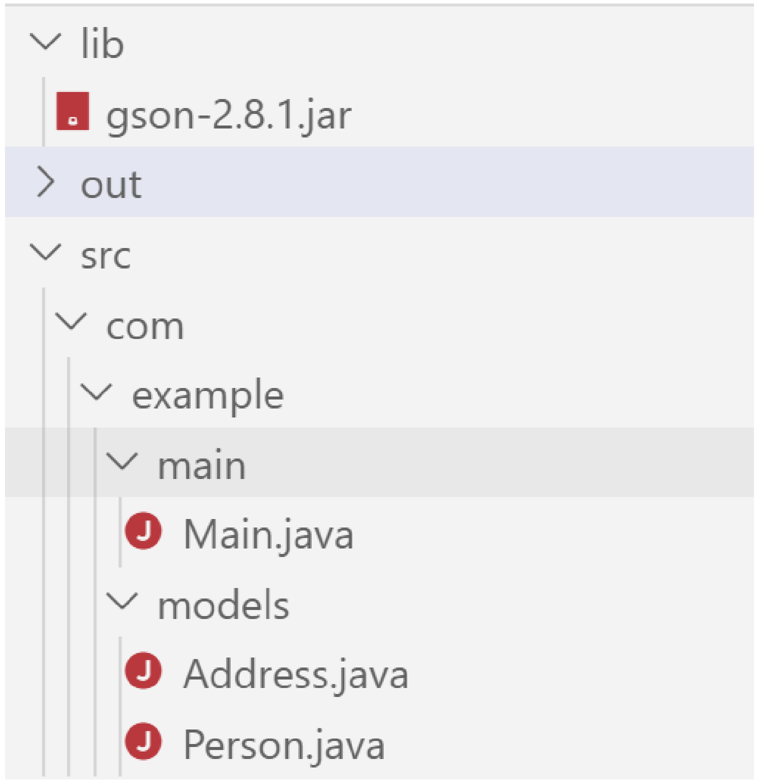
Migrating your application to Java 9

- Approach 1:
- Everything on class-path
  - Unnamed Modules
    - In Java 9 all classes found on the class-path will be included in what Java calls the unnamed module.
    - The unnamed module exports all its packages. However, the classes in the unnamed module are only readable by other classes in the unnamed module. No named module can read the classes of the unnamed module.



Migrating your application to Java 9

- Approach 1: Unnamed modules



```
lib
  gson-2.8.1.jar
out
src
  com
    example
      main
        Main.java
      models
        Address.java
        Person.java
```

```
$ jdeps -summary lib/gson-2.8.1.jar
gson-2.8.1.jar -> java.base
gson-2.8.1.jar -> java.sql
```

Compile:
Windows:

```
dir /S /B src\*.java > sources.txt
javac --class-path lib\gson-2.8.1.jar   -d out @sources.txt
jar --create --file lib\person.jar  -C out  .
```

Mac:

```
javac --class-path lib/gson-2.8.1.jar   -d out $(find src -name '*.java')
jar --create --file lib/person.jar  -C out  .
```

Run:

```
java --class-path lib\gson-2.8.1.jar;lib\person.jar  com.example.main.Main
```

Mac: use : instead of ;

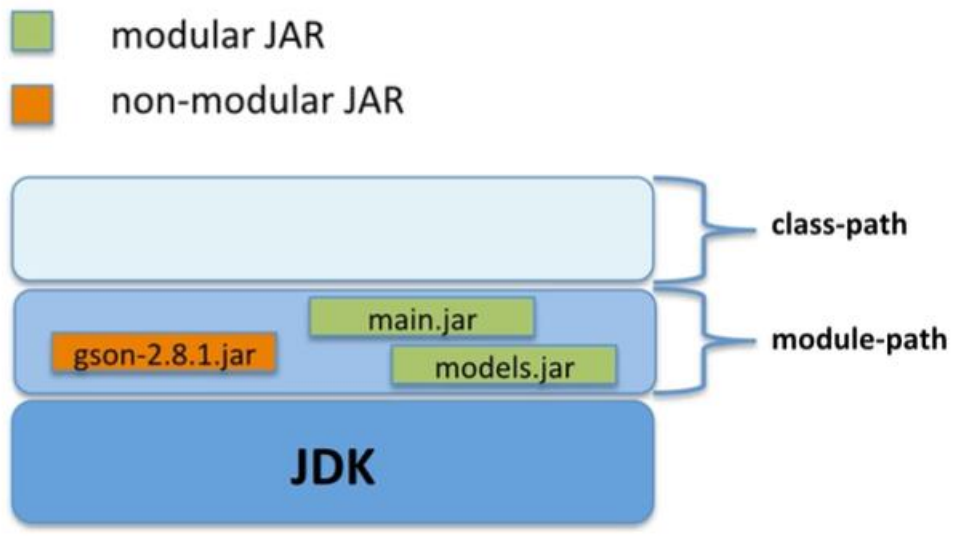
Migrating your application to Java 9

- Approach 1: Unnamed modules

```
jdeps  --class-path lib\gson-2.8.1.jar lib\person.jar
person.jar -> lib\gson-2.8.1.jar
person.jar -> java.base
  com.example.main           -> com.example.models            person.jar
  com.example.main           -> com.google.gson               gson-2.8.1.jar
  com.example.main           -> java.io                       java.base
  com.example.main           -> java.lang                     java.base
  com.example.main           -> java.util                     java.base
  com.example.models         -> com.google.gson.annotations   gson-2.8.1.jar
  com.example.models         -> java.lang                     java.base
  com.example.models         -> java.util                     java.base
```

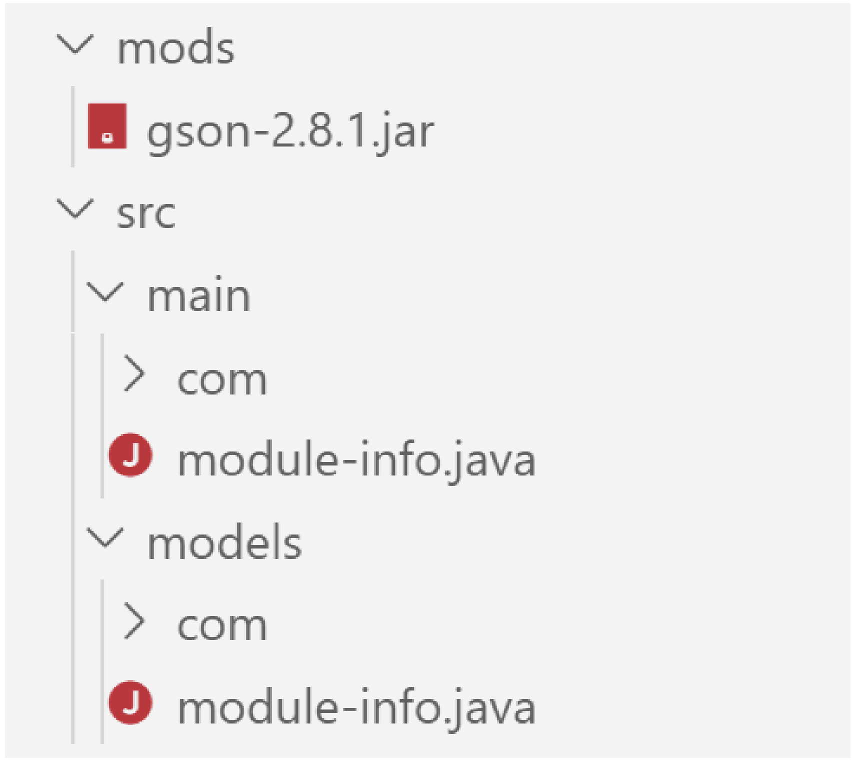
Migrating your application to Java 9

- **Automatic Modules**
    - Solution if your code is modularized but 3rd party libraries are not.
    - An *automatic module* is made from a JAR file with Java classes that are not modularized, meaning the JAR file has no module descriptor

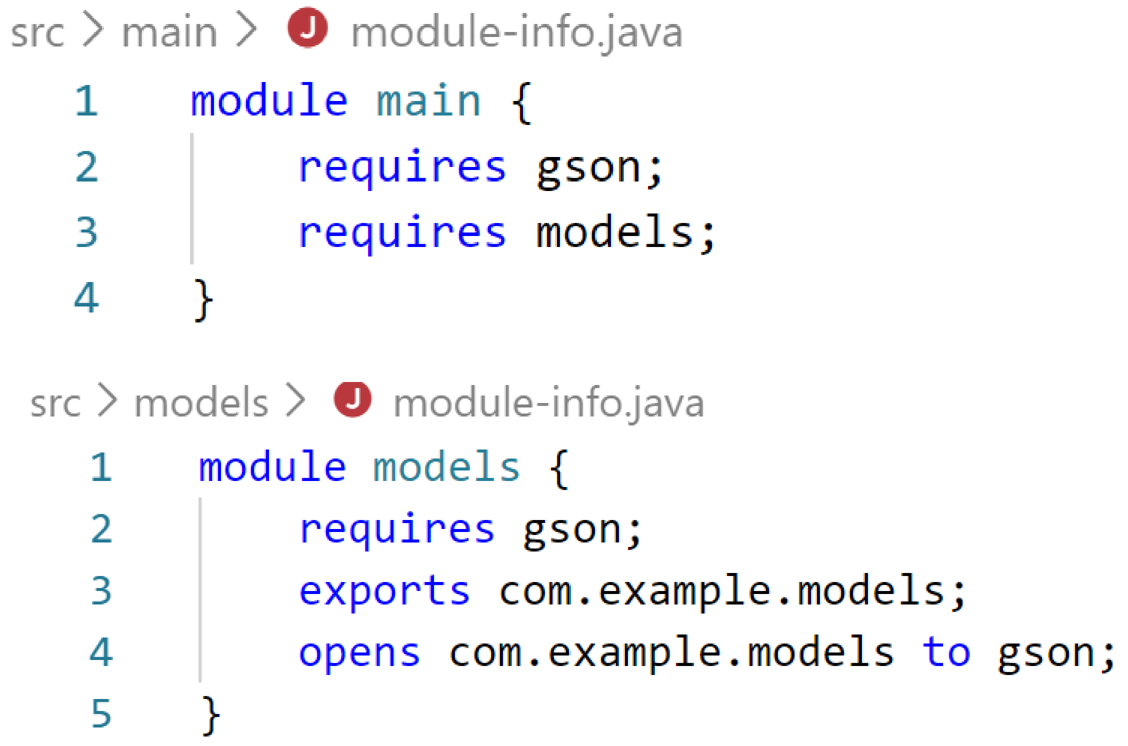


Migrating your application to Java 9

- **Automatic Modules**



```
mods
    gson-2.8.1.jar
src
    main
        com
        module-info.java
    models
        com
        module-info.java
```



src > main > module-info.java

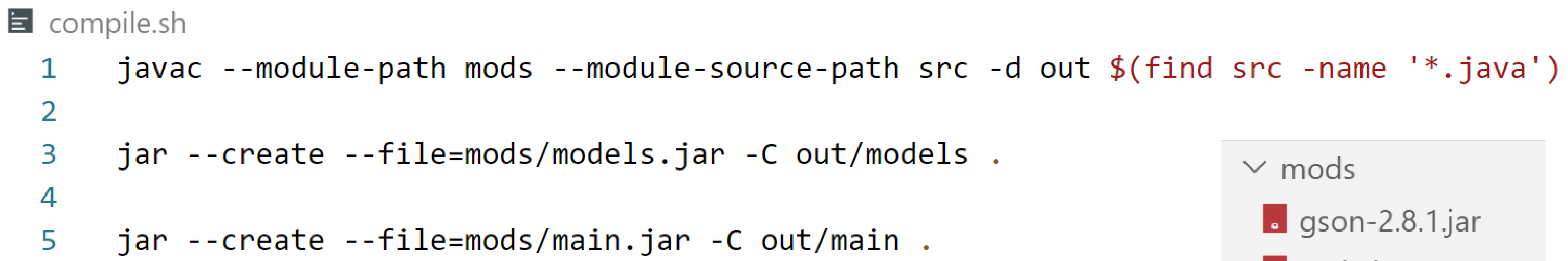
```java
module main {
    requires gson;
    requires models;
}
```

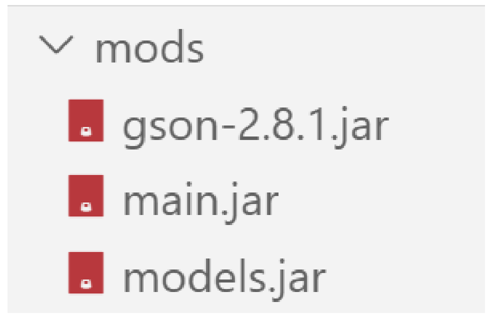
src > models > module-info.java

```java
module models {
    requires gson;
    exports com.example.models;
    opens com.example.models to gson;
}
```

Migrating your application to Java 9

- **Automatic Modules**

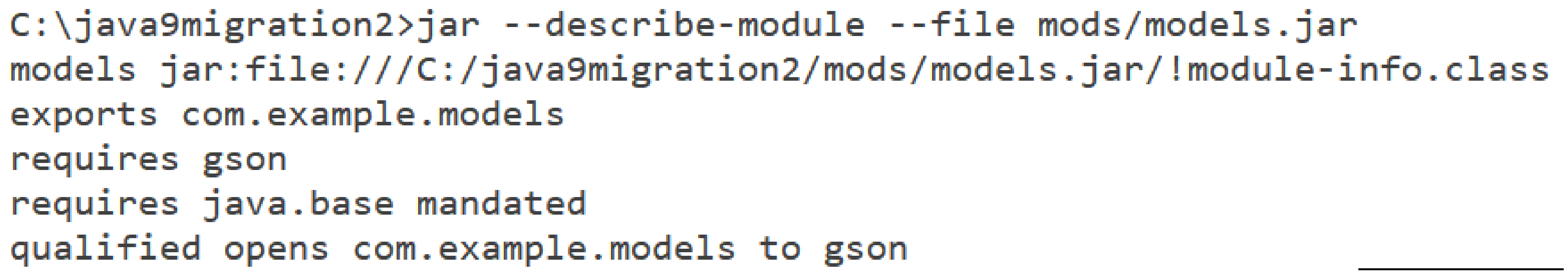

compile.sh
```
javac --module-path mods --module-source-path src -d out $(find src -name '*.java')

jar --create --file=mods/models.jar -C out/models .

jar --create --file=mods/main.jar -C out/main .
```


```
mods
  gson-2.8.1.jar
  main.jar
  models.jar
```

- **gson jar present in “mods” will be automatic module**

Migrating your application to Java 9

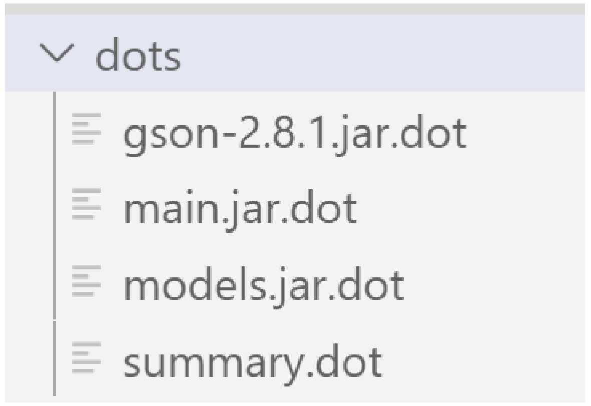
- **Automatic Modules**
- Run the application
  - java --module-path mods --add-modules java.sql --module main/com.example.main.Main
  - The java.sql module which is required for gson has to be explicitly added



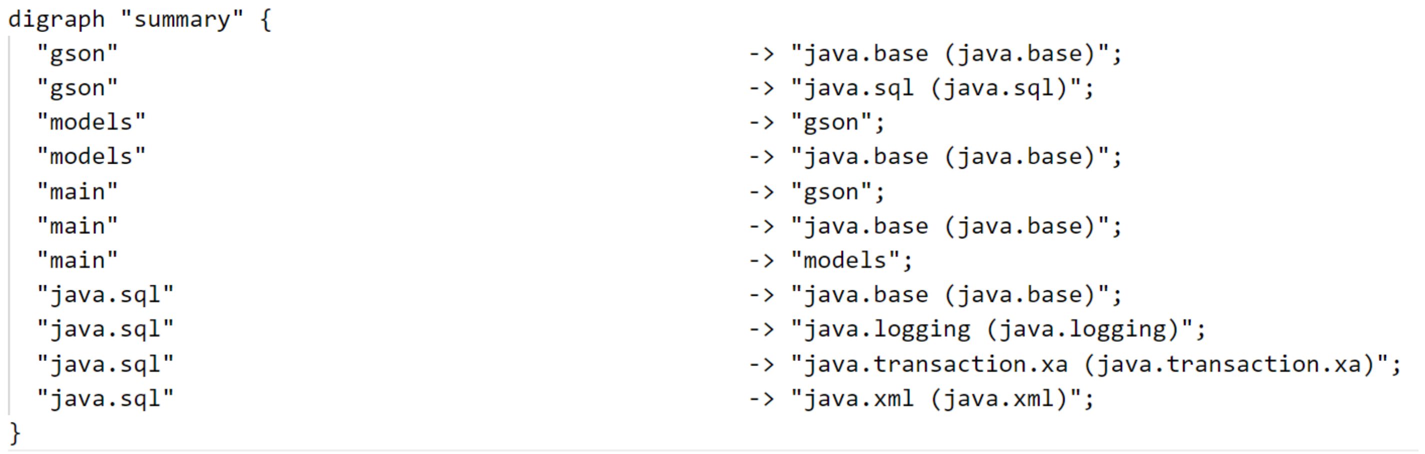
```
C:\java9migration2>jar --describe-module --file mods/models.jar
models jar:file:///C:/java9migration2/mods/models.jar/!module-info.class
exports com.example.models
requires gson
requires java.base mandated
qualified opens com.example.models to gson
```

Generate Graph

- jdeps --module-path mods --add-modules java.sql --recursive --dot-output dots mods/main.jar

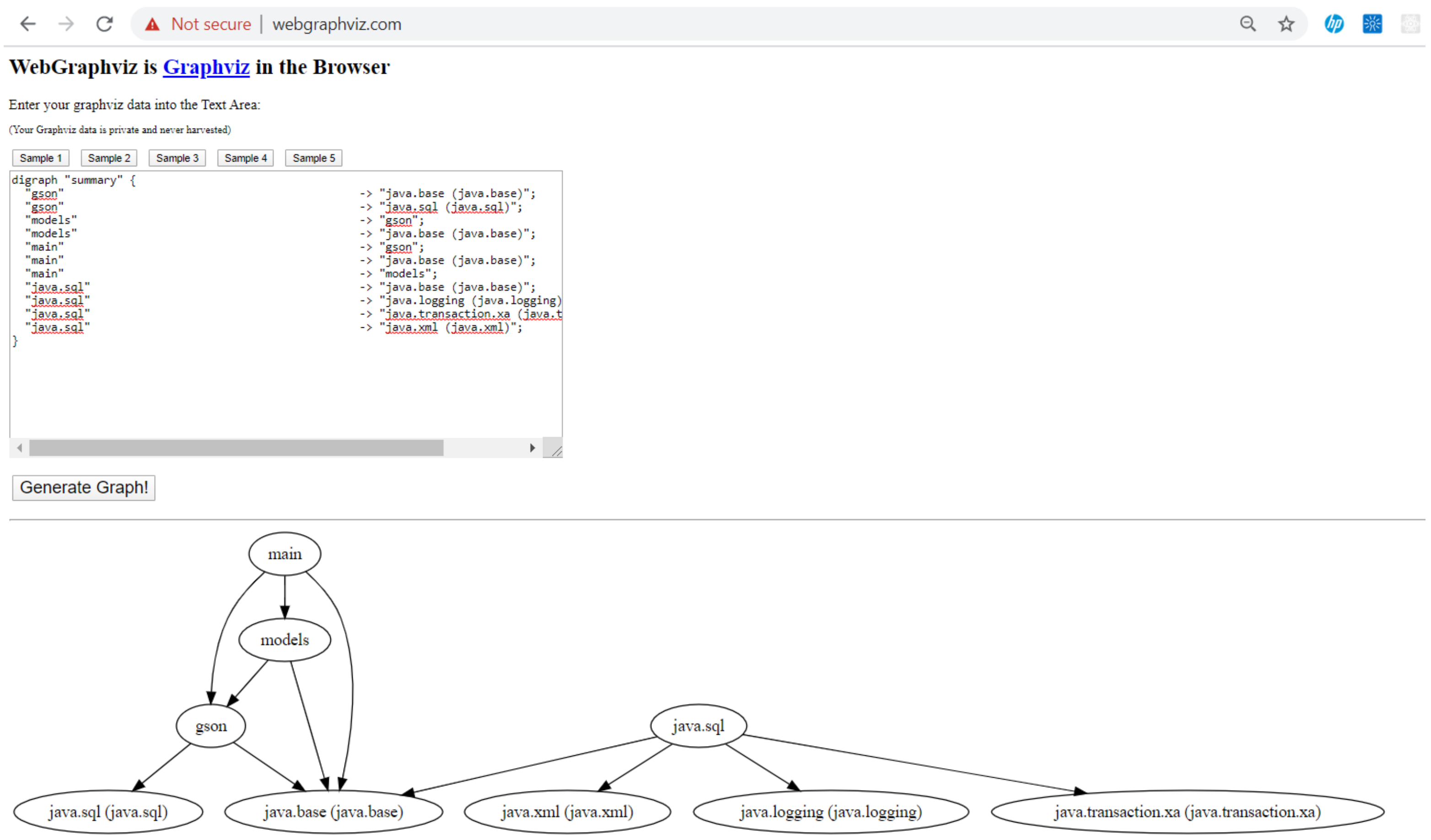


```
dots
    gson-2.8.1.jar.dot
    main.jar.dot
    models.jar.dot
    summary.dot
```



```
digraph "summary" {
  "gson"                                                  -> "java.base (java.base)";
  "gson"                                                  -> "java.sql (java.sql)";
  "models"                                                -> "gson";
  "models"                                                -> "java.base (java.base)";
  "main"                                                  -> "gson";
  "main"                                                  -> "java.base (java.base)";
  "main"                                                  -> "models";
  "java.sql"                                              -> "java.base (java.base)";
  "java.sql"                                              -> "java.logging (java.logging)";
  "java.sql"                                              -> "java.transaction.xa (java.transaction.xa)";
  "java.sql"                                              -> "java.xml (java.xml)";
}
```

Generate Graph



WebGraphviz is Graphviz in the Browser

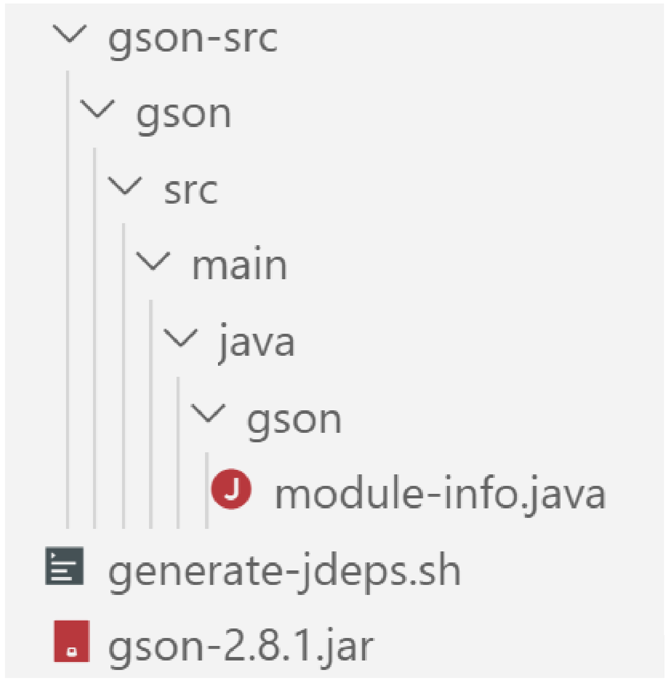
Enter your graphviz data into the Text Area:

(Your Graphviz data is private and never harvested)

```
digraph "summary" {
  "gson"                                            -> "java.base (java.base)";
  "gson"                                            -> "java.sql (java.sql)";
  "models"                                          -> "gson";
  "models"                                          -> "java.base (java.base)";
  "main"                                            -> "gson";
  "main"                                            -> "java.base (java.base)";
  "main"                                            -> "models";
  "java.sql"                                        -> "java.base (java.base)";
  "java.sql"                                        -> "java.logging (java.logging)
  "java.sql"                                        -> "java.transaction.xa (java.t
  "java.sql"                                        -> "java.xml (java.xml)";
}
```

Migrating your application to Java 9

- Completely modular application
  - Modularize the non-modular jars
    - jdeps --generate-module-info gson-src/gson/src/main/java gson-2.8.1.jar



```
gson-src
  gson
    src
      main
        java
          gson
            module-info.java
generate-jdeps.sh
gson-2.8.1.jar
```

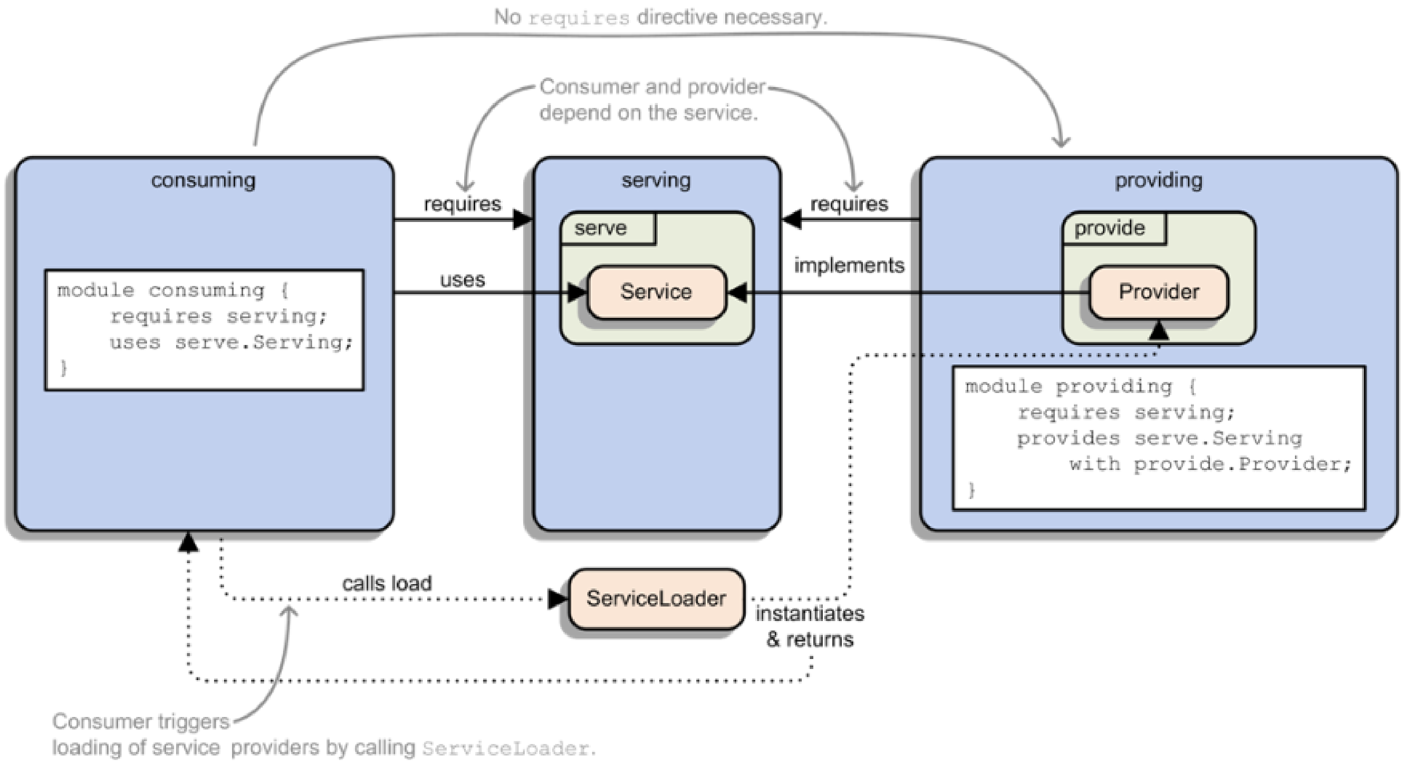
```
module gson {
    requires transitive java.sql;

    exports com.google.gson;
    exports com.google.gson.annotations;
    exports com.google.gson.internal;
    exports com.google.gson.internal.bind;
    exports com.google.gson.internal.bind.util;
    exports com.google.gson.reflect;
    exports com.google.gson.stream;

}
```

Migrating your application to Java 9

- Modularize the non-modular jars
  - Extract java files from gson-2.8.1.jar
  - javac --source-path gson-src/gson/src/main/java -d out/gson $(find gson-src/gson/src/main/java -name *.java)
  - jar --create --file=mods/gson.jar –C out/gson .
  - The gson.jar file created is modularized now

Services in the Java Platform Module System

- A service is an accessible type that one module wants to use and another module provides an instance of.
- At run time, the depending module can use the **ServiceLoader** class to get all provided implementations of a service by calling **ServiceLoader.load(${service}.class)**
- The module system will then return a Provider<${service}> for each provider that any module in the module graph declares.


No requires directive necessary.
Consumer and provider depend on the service.
consuming
requires
uses
serving
serve
Service
requires
implements
providing
provide
Provider
calls load
ServiceLoader
instantiates & returns
Consumer triggers loading of service providers by calling ServiceLoader.

```
module consuming {
    requires serving;
    uses serve.Serving;
}
```

```
module providing {
    requires serving;
    provides serve.Serving
        with provide.Provider;
}
```

Refer book-app.zip

Running Maven On Java 9

- **Maven Version**
  - **Maven itself**: 3.5.0
  - **Maven Compiler Plugin**: 3.7.0 / 3.8.0 for Java 11

```
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>11</source>
          <target>11</target>
          <showWarnings>true</showWarnings>
          <showDeprecation>true</showDeprecation>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

jlink: The Java Linker

- JLink
  - A Java command line driven tool which links / brings together all required modules for an application into a run time image.
- This image is usually drastically smaller in size, thus helping reduce the footprint of the application as the entire JRE is not normally needed.
- jlink will also resolve static dependencies between modules thus guaranteeing the integrity of each module at run time.
- jlink requires that all artifacts are modules with well defined public contracts (exports, requires etc), thus “Automatic” modules will not suffice.

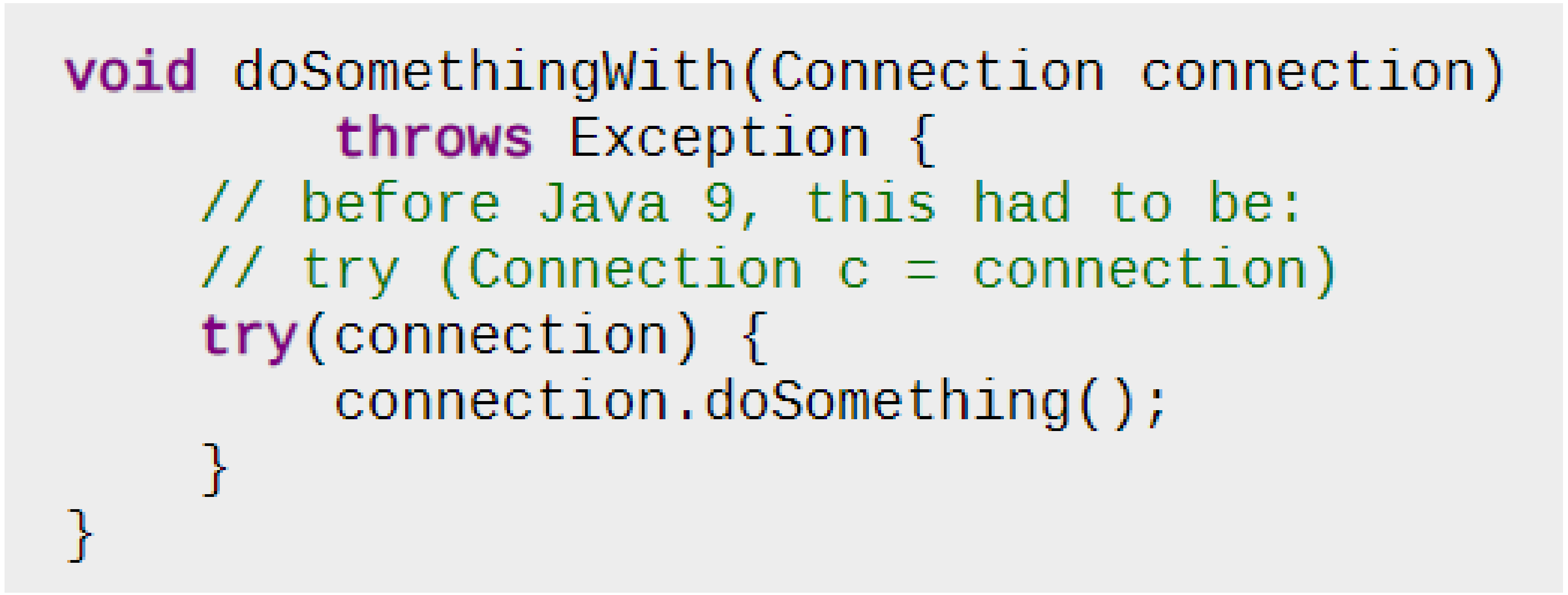
Java 9 Private Interface methods

- They are exactly like other private methods:
  - can not be abstract
  - Can't override

```
public interface InJava8 {

    default boolean evenSum(int... numbers) {
        return sum(numbers) % 2 == 0;
    }

    default boolean oddSum(int... numbers) {
        return sum(numbers) % 2 == 1;
    }

    // before Java 9, this had to be `default`
    // and hence public
    private int sum(int[] numbers) {
        return IntStream.of(numbers).sum();
    }

}
```

Java 9 :Try With Effectively Final Resources

- If connection is effectively final, you can write **try** (connection) instead of the **try** (Connection c = connection) that you had to use before Java 9.



```
void doSomethingWith(Connection connection)
        throws Exception {
    // before Java 9, this had to be:
    // try (Connection c = connection)
    try(connection) {
        connection.doSomething();
    }
}
```

Java 9 Diamond Operator Improvement

- **Diamond Operator**
  - // since Java 7, diamond operator can be used
    List<String> myList = new ArrayList<>();
  - Unfortunately, the use of the Diamond Operator was not possible for anonymous inner classes.

  - Java 9, diamond operator can be used for anonymous inner classes.
  -

```
Predicate<Integer> predicate = new Predicate<>() {
    @Override
    public boolean test(Integer input) {
        return input == 42;
    }
};
```

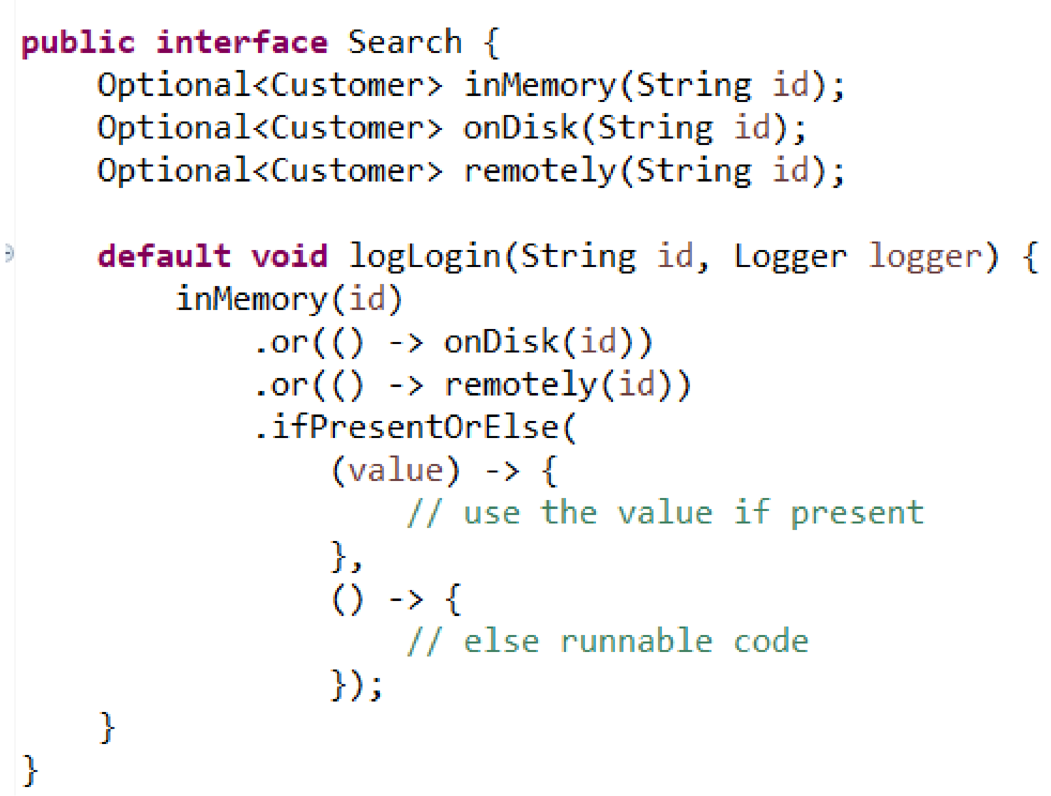
Collection Factories

- The new collection factory methods List::of, Set::of, Map::of return collections that:
    - are immutable (unlike e.g. **Array**::asList, where elements can be replaced) but do not express that in the type system – calling e.g. List::add causes an UnsupportedOperationException
    - roundly reject **null** as elements/keys/values (unlike ArrayList, HashSet, and HashMap, but like ConcurrentHashMap)

```
List<String> list = List.of("a", "b", "c");
Set<String> set = Set.of("a", "b", "c");
Map<String, Integer> mapImmediate = Map.of(
    "one", 1,
    "two", 2,
    "three", 3);
Map<String, Integer> mapEntries = Map.ofEntries(
    entry("one", 1),
    entry("two", 2),
    entry("three", 3));
```

Java 9 Additions to Optional

- Optional::**or** takes a supplier of another Optional and when empty, returns the instance supplied by it; otherwise returns itself.
- Optional::ifPresentOrElse extends Optional ::isPresent to take an additional parameter, a Runnable, that is called if the Optional is empty.



```java
public interface Search {
    Optional<Customer> inMemory(String id);
    Optional<Customer> onDisk(String id);
    Optional<Customer> remotely(String id);

    default void logLogin(String id, Logger logger) {
        inMemory(id)
            .or(() -> onDisk(id))
            .or(() -> remotely(id))
            .ifPresentOrElse(
                (value) -> {
                    // use the value if present
                },
                () -> {
                    // else runnable code
                });
    }
}
```

Java 9 Additions to Optional

- **Optional::stream**
  - Optional::stream creates a stream of either zero or one element, depending on the optional is empty or not – great to replace .filter(Optional::isPresent).map(Optional::get) stream pipelines with .flatMap(Optional::stream).

  - Java 8 way

  Given a method Optional<Customer> findCustomer(String customerId) we had to do something like this:

```
public Stream<Customer> findCustomers(Collection<String> customerIds) {
        return customerIds.stream()
                        .map(this::findCustomer)
                        // now we have a Stream<Optional<Customer>>
                        .filter(Optional::isPresent)
                        .map(Optional::get);
}
```

Java 9 Additions to Optional

- Java 9 with Optional::stream

```
public Stream<Customer> findCustomers(Collection<String> customerIds) {
        return customerIds.stream()
                .map(this::findCustomer)
                .flatMap(Optional::stream)
}
 OR
public Stream<Customer> findCustomers(Collection<String> customerIds) {
        return customerIds.stream()
        .flatMap(id -> findCustomer(id).stream());
}
```

HTTP/2 Client in Java 9

- Advantage of HTTP/2
  - HTTP/2 can send multiple requests for data in parallel over a single TCP connection. This is the most advanced feature of the HTTP/2 protocol because it allows you to download web files via ASync mode from one server
  - It allows servers to “push” responses proactively into client caches instead of waiting for a new request for each resource.
  - It reduces additional round trip times (RTT), making our website load faster without any optimization.

HTTP/2 Client in Java 9

- There are 3 new classes introduced to handle HTTP communication.
    - HttpClient
        - HttpClient handles the creation and send of requests.
    - HttpRequest
        - HttpRequest is used to construct a request to be sent via the HttpClient.
    - HttpResponse
        - HttpResponse holds the response from the request that has been sent.

HTTP/2 Client in Java 9

```
HttpClient httpClient = HttpClient.newHttpClient(); // Create a HttpClient

HttpRequest httpRequest = HttpRequest.newBuilder()
        .uri(new URI("https://jsonplaceholder.typicode.com/todos")).GET().build();

System.out.println("Calling...");
    CompletableFuture<HttpResponse<String>> httpResponse = httpClient.sendAsync(httpRequest, HttpResponse.BodyHandler.asString());
CompletableFuture<HttpResponse<String>> httpResponse = httpClient.sendAsync(httpRequest,
        HttpResponse.BodyHandlers.ofString()); // Send the request asynchronously

System.out.println("Performing Other Task");

System.out.println(httpResponse.get().body());
```

Local-Variable Type Inference

- From Java 10 on developers can choose to let the compiler infer types by using var
  - Note: var is statically typed unlike JavaScript
- Motivation
  - Developers frequently complain about the degree of boilerplate coding required in Java

```
var i = 42;         //int
var s = '42';       //String
var list1 = new ArrayList();            //ArrayList of Objects;
var list2 = new ArrayList<String>();    //ArrayList of Strings
```

Local-Variable Type Inference

It definitely saves typing and makes the code less cluttered with repeated code. Let's look at this example:

```
Map<Integer, List<String>> idToNames = new HashMap<>();
//...
for(Map.Entry<Integer, List<String>> e: idToNames.entrySet()){
    List<String> names = e.getValue();
    //...
}
```

That was the only way to implement such a loop. But since Java 10, it can be written as follows:

```
var idToNames = new HashMap<Integer, List<String>>();
//...
for(var e: idToNames.entrySet()){
    var names = e.getValue();
    //...
}
```

Local-Variable Type Inference

```java
interface A {
    void m();
}

static class AImpl implements A {
    public void m(){}
    public void f(){}
}
```

```java
A a1 = new AImpl();
a1.m();
//a1.f();  //does not compile

var a = new AImpl();
a.m();
a.f();
```

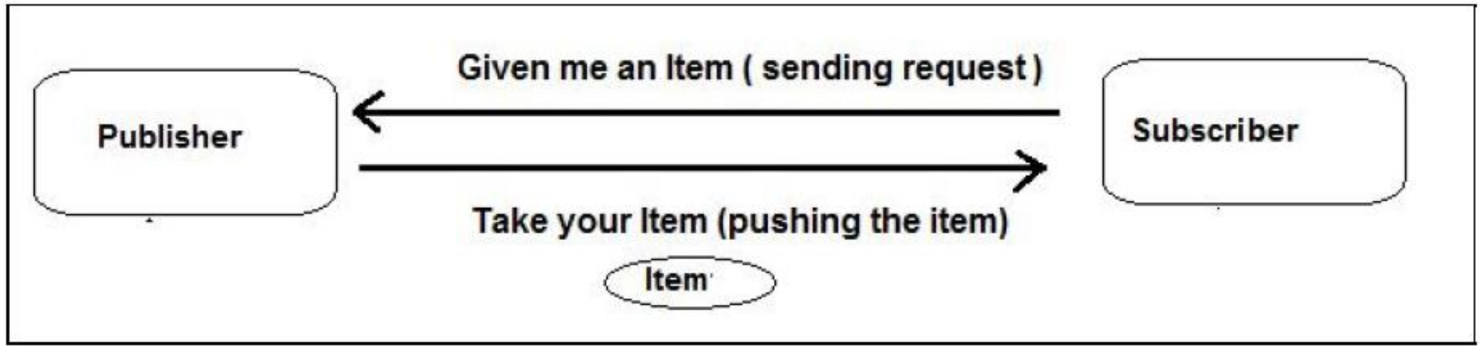
Local-Variable Type Inference

- **Where Not To use Var**

```
// nope
var foo;
foo = "Foo";
// It really has to be var foo = "Foo".

//var won’t work with so called “poly expressions”,
// like lambdas and method references
// none of this works
var ints = {0, 1, 2};
var appendSpace = a -> a + " ";
var compareString = String::compareTo

var users = new ArrayList<User>();
// this is a compile error:
users = new LinkedList<>();
```

Reactive Programming in Java 9

- Java 9 comes up with Reactive Streams handled by Flow APIs supporting Publisher- Subscriber pattern.
- The **Subscriber** will send a request for the number of items.
- The **Publisher** will now push the requested number of items to the **Subscriber**. **Publisher**-**Subscriber** is a bidirectional flow, where the **Publisher** emits the data and the **Subscriber** utilizes the data.
- The demand for an item has been done asynchronously.
- It is neither a pull-based nor a push-based system; rather it's a hybrid of these two.
- When the **Subscriber** is faster than the **Publisher** it works as push-based and if the **Subscriber** is slow it acts like a pull-based.


Given me an Item (sending request)
Publisher
Subscriber
Take your Item (pushing the item)
Item

Reactive Programming in Java 9

- The Flow API has the following four major components acting as the backbone of the system:
  - Publisher
  - Subscriber
  - Processor
  - Subscription

The SubmissionPublisher class

- The SubmissionPublisher class is an implementation of the Publisher interface whose role is to emit the data.
- The SubmissionPublisher class uses the supplied Executor, which delivers to the Subscriber.
- For each Subscriber, the SubmissionPublisher class uses an independent buffer whose size can be specified at the time of instance creation.
- These buffers get created on the very first use and eventually expanded to their maximum size as required

Subscriber

- It has the methods for demanding and subscribing the data.
- It also has the method to handle an error and the completion of items.

Name of callback method	**Method description**
`onSubscribe`	This method will be invoked before invoking any other method on the `Subscriber` for a particular subscription
`onNext`	This method will be invoked with a subscription's next item.
`onError`	This is invoked when any unrecoverable error has occurred either in `Publisher` or `Subscriber`. Once the method is invoked, further invocation of the methods of the `Subscriber` will be stopped.
`onComplete`	This method is invoked when all the messages have been issued to the `Subscriber` and the `Subscriber` receives the `onComplete` event.

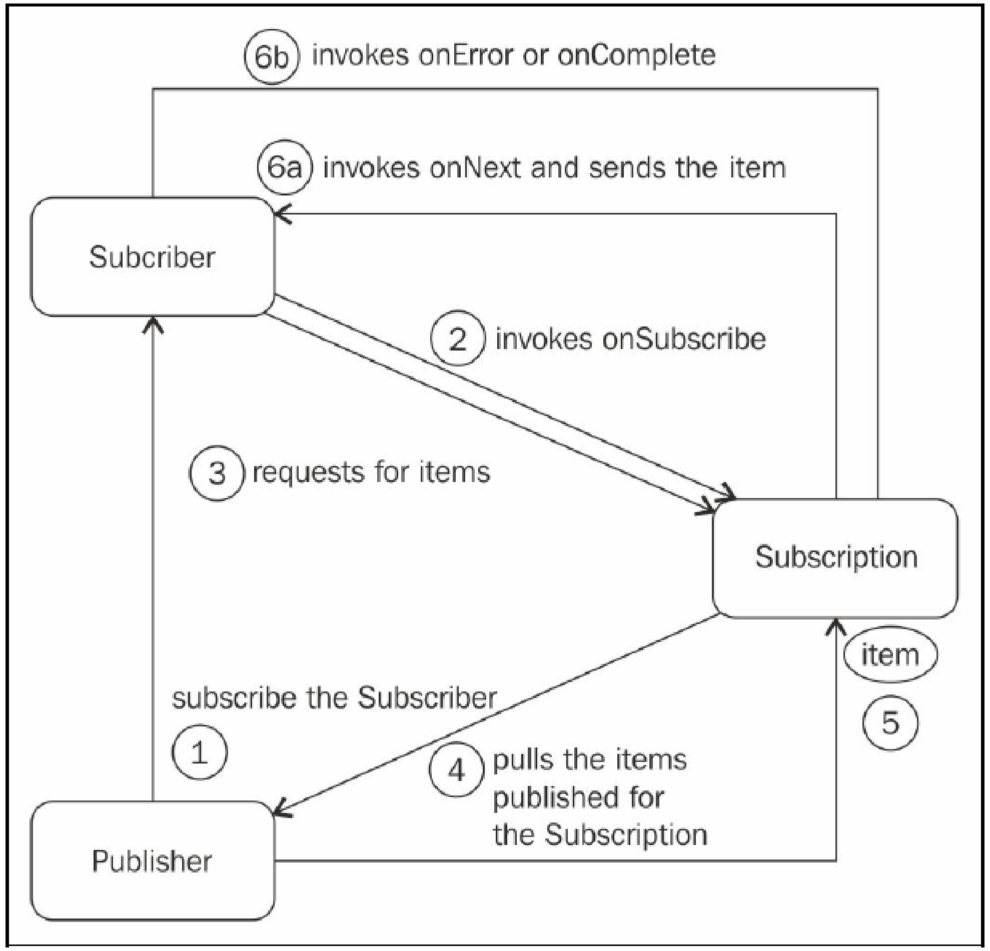
Subscription

- Subscriber subscribes the data through Subscription.

Name of callback method	**Method description**
`request`	This method adds the given number of items to the demand that is not yet fulfilled for the particular subscription. There are *n* number of requested elements. The value of *n* has to be a positive integer number that is greater than zero, otherwise, an `onError()` signal is sent to the `Subscriber`.
`cancel`	Receiving the messages will be stopped by the `cancel()` method. If the subscription is cancelled, `onComplete` or `onError` signals will not be received.

Publisher-Subscriber system

- The **Publisher** uses the subscribe() method denoting that the **Subscriber** can now subscribe the **Publisher**.
- The **Subscriber** subscribes the **Publisher** by invoking onSubscribe(). The **Subscriber** now has **Subscription**. Each **Publisher** defines its own **Subscription** and also has the logic to produce the data for the respective **Subscription**.
- The **Subscriber** uses the **Subscription** and requests for the items. The **Subscription** acts as the link between the **Publisher** and the **Subscriber**.
- The request() method invokes the onNext() method of the **Subscriber** for sending the data.


6b invokes onError or onComplete
6a invokes onNext and sends the item
Subcriber
2 invokes onSubscribe
3 requests for items
Subscription
item
5
subscribe the Subscriber
1
4 pulls the items published for the Subscription
Publisher