Workshop
Java Modularization

https://github.com/Michaeli71/JAX-London-Best-of-Java9-17.git

Michael Inden

Independent SW Consultant and Author

Speaker Intro

- Michael Inden, Year of Birth 1971
- Diploma Computer Science, C.v.O. Uni Oldenburg

- ~8 ¼ Years **SSE** @ Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Years **TPL / SA** @ IVU Traffic Technologies AG in Aachen
- ~4 ¼ Years **LSA / Trainer** @ Zühlke Engineering AG in Zürich
- ~3 Years **TL / CTO** @ **Direct Mail Informatics / ASMIQ** in Zürich
- **Freelance and Conference Speaker and Trainer**
- Author @ dpunkt.verlag

E-Mail: michael.inden@asmiq.ch
Blog: https://jaxenter.de/author/minden

https://github.com/Michaeli71/JAX-London-Best-of-Java9-17.git

Agenda

Workshop Contents

- **Modularization with Project Jigsaw**
  - PART 1: Introduction
  - PART 2: Visiblity and transitive dependencies
  - PART 3: Decouple dependencies using services
  - PART 4: Inclusion of external modules / migration strategies
- **Conclusion**

PART 1: Introduction

Introduction

Let's clarify the following questions:

- **Why modularization? What are the main goals?**
- **What is a module?**
- **Why and how to modularize the JDK?**
- **How to manage dependencies and what improved for visibilities?**

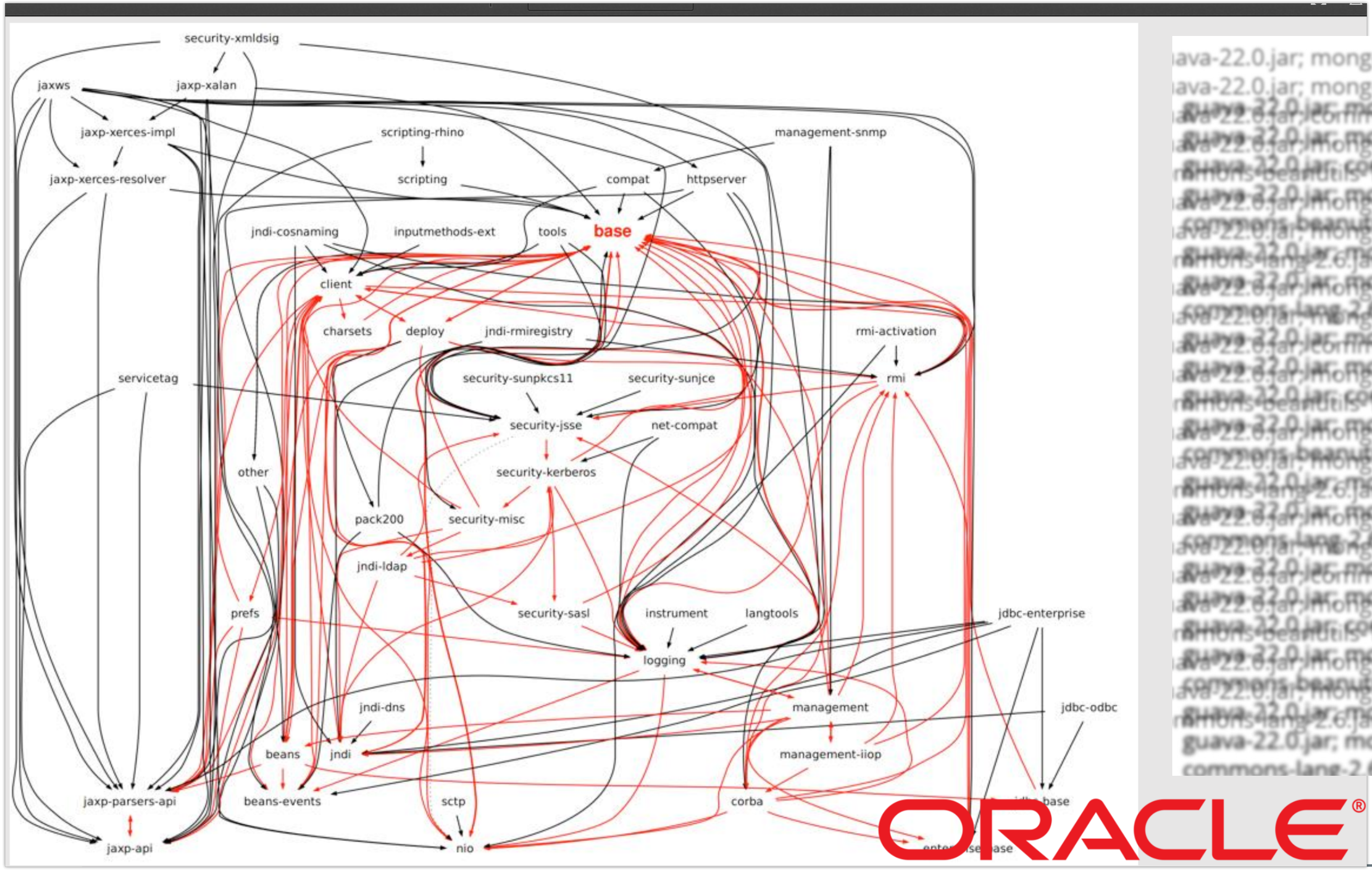
Introduction: Main goals of modularization

- **Modularization of applications and libraries**
- **Modularization of JDK itself**
- **Reliable configuration instead of error prone dependency management**

Why is modularization desirable?

Introduction: Reasons for modularization

- **Situation with Java 8: Confusing dependencies and chaos in the CLASSPATH**



http://baselone.ch/wp-content/uploads/sites/7/2017/10/Java9.pdf

Introduction: Modules in Java 8

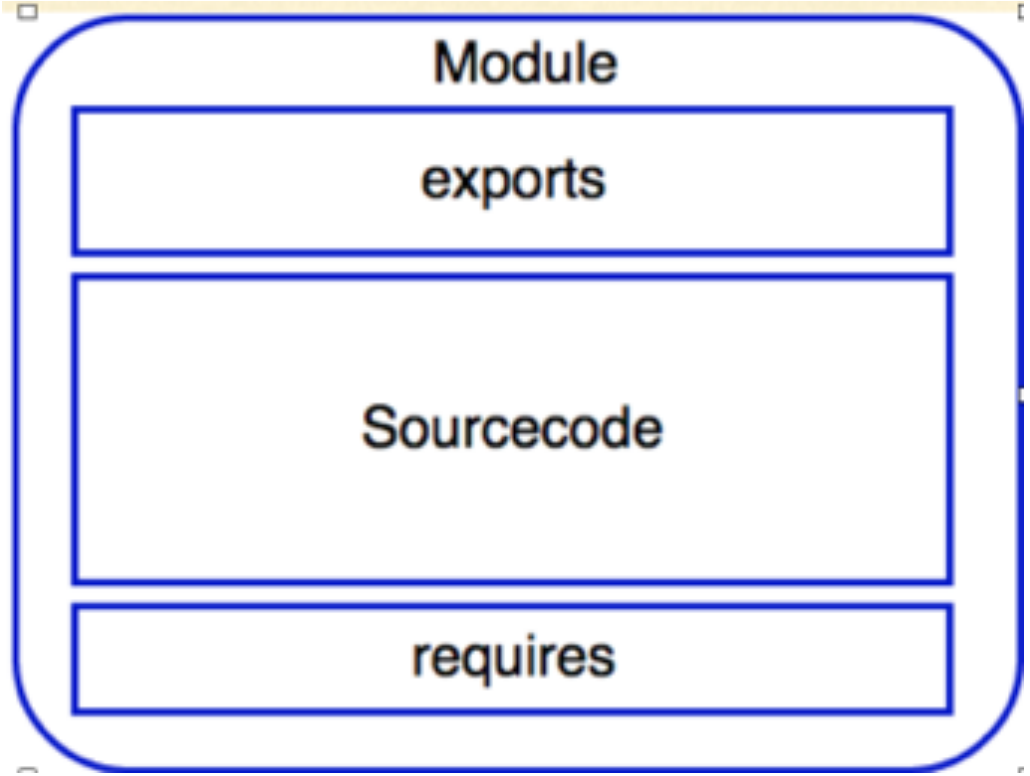
Situation with Java 8:

- **"Modules" as a collection of classes in packages or as JARs**
  - loose coupling cannot be forced
  - Dependencies and visibility difficult to control
- **"Modules" based on OSGi**
  - very mature
  - but somewhat complex and not suitable for the JDK

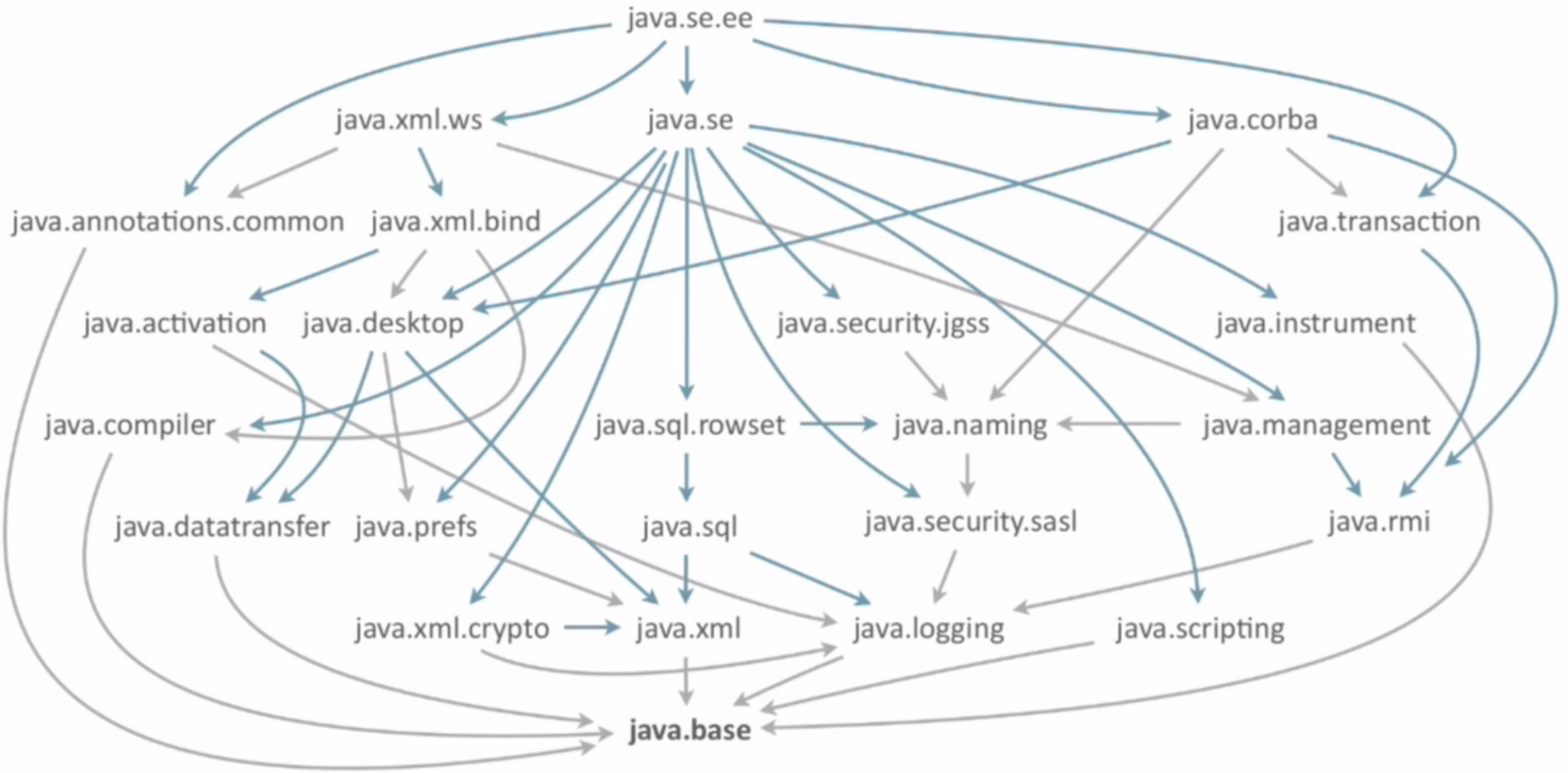
Introduction: Modules in Java 9

- **Modules as new software build blocks in the JDK for grouping packages**

- **One module ...**
  - has an unique name
  - consists of packages, classes, and so on.
  - offers limited access to functionality
  - hides implementation details
  - defines all dependencies
  - has a well-defined interface



Introduction: Modularization of the JDK

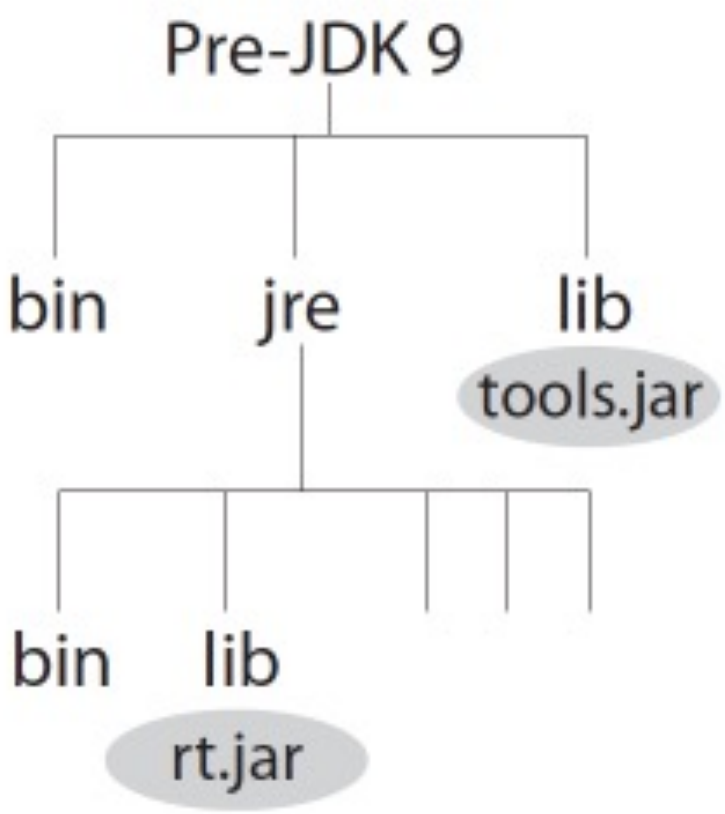


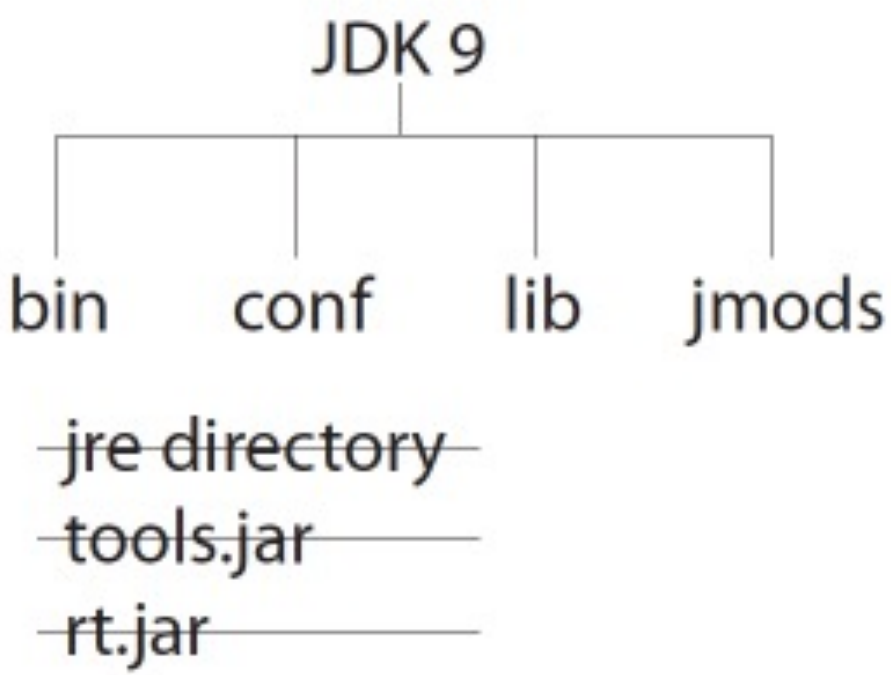
Modules in One Lesson by Mark Reinhold – https://www.youtube.com/watch?v=rFhhLXcOBsk

Introduction: Modularization of the JDK

- JDK directory structure massively changed, no distinction JDK / JRE anymore
- rt.jar and tools.jar no longer exist => modules in dir jmods
- visibility restrictions => some internal APIs no longer accessible
- Extension of JDK with "endorsed dirs" no longer supported
- Split Packages no longer supported
- There is now a linker with which you can create special executables of your own program.

Introduction: Modularization of the JDK



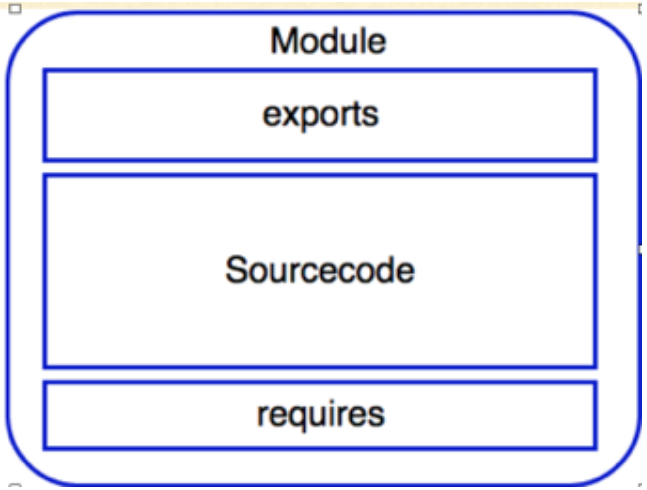


And what do modules look like?

Introduction: Modules in Java 9

- **Modules as new software build blocks in the JDK in addition to packages**
- **Each module has a module descriptor `module-info.java`**

```
module <ModuleName>
{
    requires <ModuleNameOfRequiredModule>;

    exports <PackageName>;
}
```



- **Contains a set of packages and classes**
- **Defines the number of dependencies.**
- **The module system ensures that each dependency is fulfilled exactly by one module and that the dependencies are acyclic.**

Example 2 Modules: Compiling, adapting in the Java ecosystem

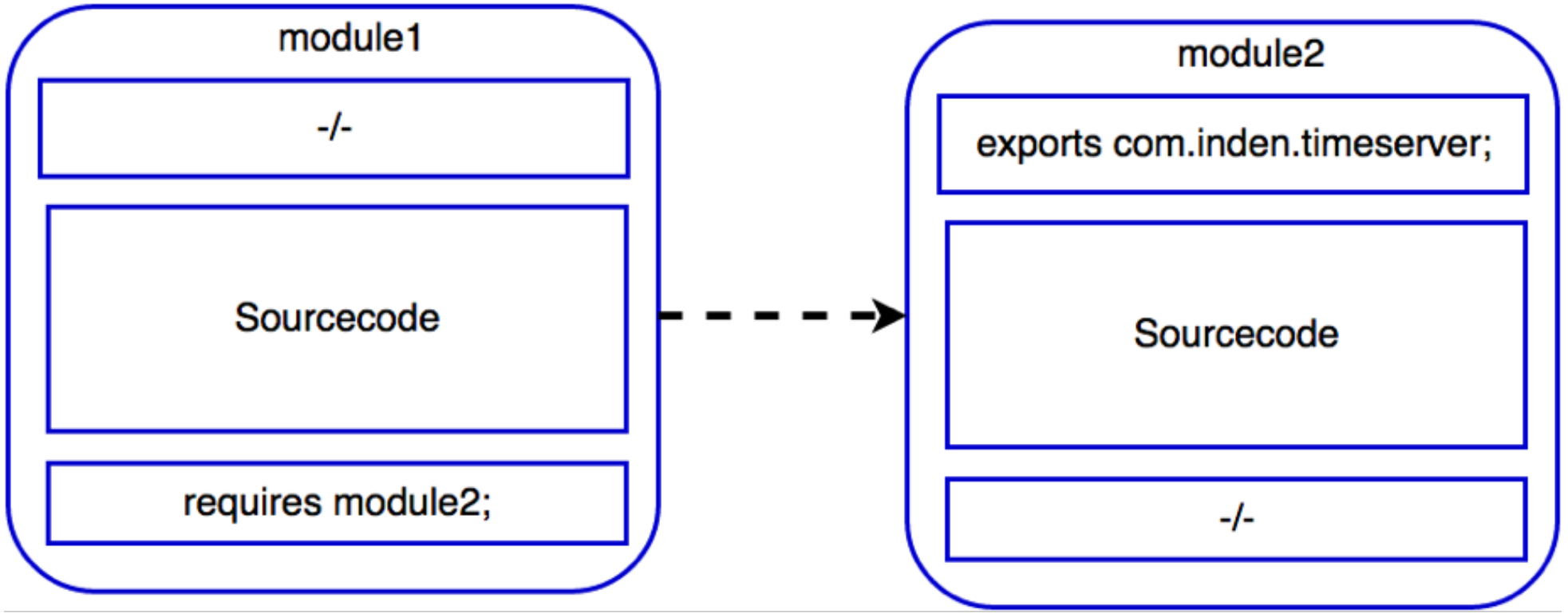
- **Modules represent an additional hierarchy for packages.**
- **Therefore extensions for tools were necessary**
- **Java compiler has been enhanced with module path specification:**

```
javac --module-path <module-path> …  oder javac –p <module-path> …
```

- **Java runtime was extended by module path and class to be started with modules:**

```
java --module-path <modulepath> -m <modulename>/<moduleclass>
```

Introduction: Modules in Java 9




```
module module1
{
    requires module2;
}
```

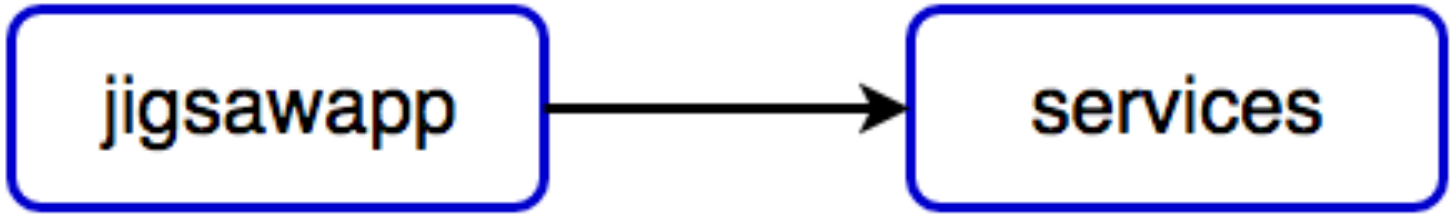
```
module module2
{
    exports com.inden.timeserver;
}
```

Introduction: Modules in Java 9: Readability / Accessibility

- **Module 1 requires Module 2 => Module 1 reads Module 2**
- **Readability is the prerequisite for module 1 to be able to reference the types from module 2.**
- **Accessibility: Readability + exports: A class A from module 2 is only accessible if module 1 reads the corresponding module 2 and module 2 additionally exports the required package. => strong encapsulation**
- **Both form the basis for a reliable configuration.**

Example using 2 Modules

Example 2 Modules



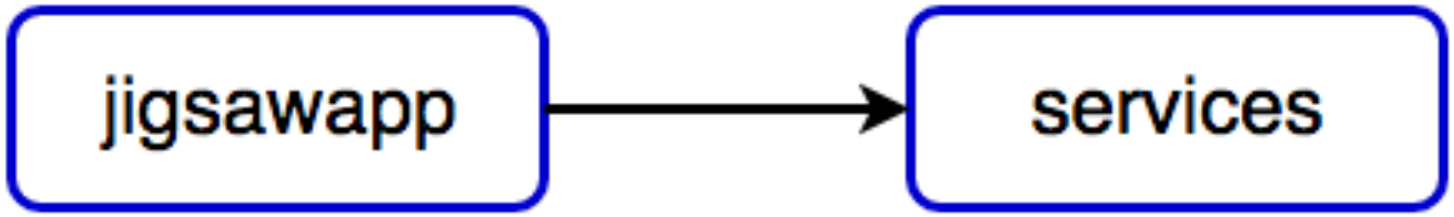
Directory layout (proposed by Oracle)

- application with several modules => one common src dir
- source code is stored in a subdirectory with the module name

```
'-- src
    |-- com.inden.module1
    |   |-- com
    |   |   '-- inden
    |   |       '-- module1
    |   |           '-- Application.java
    |   '-- module-info.java
    '-- com.inden.module2
        |-- com
        |   '-- inden
        |       '-- module2
        |           '-- OtherClass.java
        '-- module-info.java
```

- Practical only for special applications and first examples

Example 2 Modules: Dependencies and dir layout




```
└── src
    ├── jigsawapp
    │   ├── com
    │   │   └── inden
    │   │       └── javaprofi
    │   │           └── MessageExample.java
    │   └── module-info.java
    └── services
        ├── com
        │   └── services
        │       └── MessageService.java
        └── module-info.java
```

Example 2 Modules: Implementation

- **Module descriptors**

```
module jigsawapp {
    requires services;
}

module services {
    exports com.services;
}
```

- **Implementation of class for the module services**

```
package com.services;

public class MessageService
{
    public String createGreetingMessage(final String name)
    {
       return "Hello " + name;
    }
}
```

Example 2 Modules: Implementation

- **Implementation of class for the module `jigsawapp`**

```
package com.inden.javaprofi;

import com.services.MessageService;

public class MessageExample
{
    public static void main(String[] args)
    {
        var msgService = new MessageService();
        System.out.println(msgService.createGreetingMessage("Mainz"));
    }
}
```

Example 2 Modules: Compilation

- **Step 3: Compilation first for module `services`, then module `jigsawapp`**

```
javac -d build/services \
          src/services/*.java \
          src/services/com/services/*.java
```

```
javac -d build/jigsawapp \
          src/jigsawapp/*.java \
          src/jigsawapp/com/inden/javaprofi/*.java
```

=>

```
src/jigsawapp/module-info.java:2: error: module not found: services
requires services;
         ^
1 error
```

Example 2 Module: Compilation

- **Step 3: Compilation of module `jigsawapp`**

```
javac -d build(jigsawapp \
       -p build \
          src/jigsawapp/*.java \
          src/jigsawapp/com/inden/javaprofi/*.java
```

```
build
├── jigsawapp
│   ├── com
│   │   └── inden
│   │       └── javaprofi
│   │           └── MessageExample.class
│   └── module-info.class
└── services
    ├── com
    │   └── services
    │       └── MessageService.class
    └── module-info.class
```

Example 2 Modules: Compliation with Multi Module Build

- **Step 3: Compilation with Multi Module Build**
  - Für MAC und Linux:

```
javac --module-source-path src -d build $(find src -name '*.java')
```

  - Für Windows mit Powershell:

```
javac -d build --module-source-path src $(dir src -r -i '*.java')
```

```
build
├── jigsawapp
│   ├── com
│   │   └── inden
│   │       └── javaprofi
│   │           └── MessageExample.class
│   └── module-info.class
└── services
    ├── com
    │   └── services
    │       └── MessageService.class
    └── module-info.class
```

Example 2 Modules: Packaging and start of application

- **Step 4: Packaging**

```
mkdir lib

jar --create --file lib/services.jar -C build/services .

jar --create --file lib/jigsawapp.jar -C build/jigsawapp .
```

- **Step 5: Start of application**

```
java -p lib -m jigsawapp/com.inden.javaprofi.MessageExample
```

Example 2 Modules: Packaging and start of application

- **Step 4: Packaging**

```
mkdir lib

jar --create --file lib/services.jar -C build/services

jar --create --file lib/jigsawapp.jar \
    --main-class com.inden.javaprofi.MessageExample \
    -C build/jigsawapp .
```

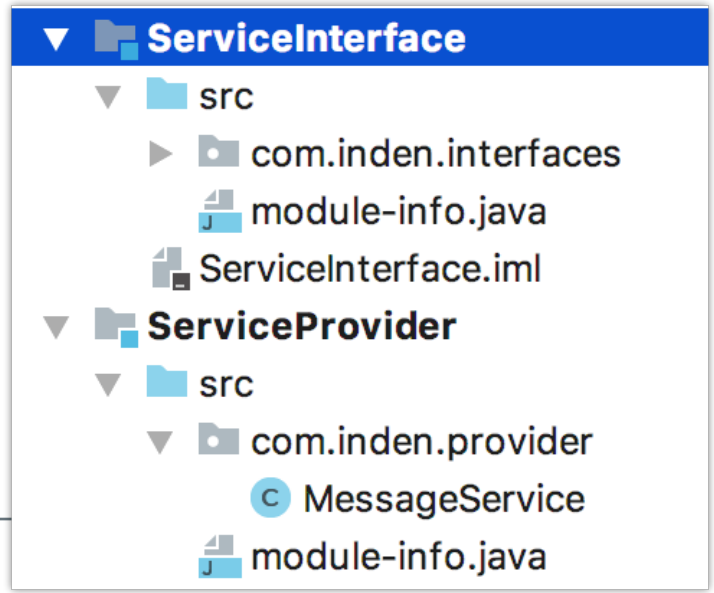
- **Step 5: Start of application**

```
~~java -p lib -m jigsawapp/com.inden.javaprofi.MessageExample~~
java -p lib -m jigsawapp
```

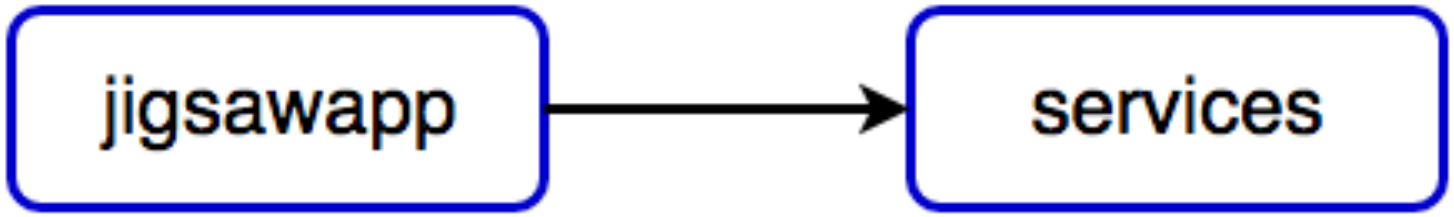
Modules as high-level building blocks, but then we have to act as low-level console junkie?

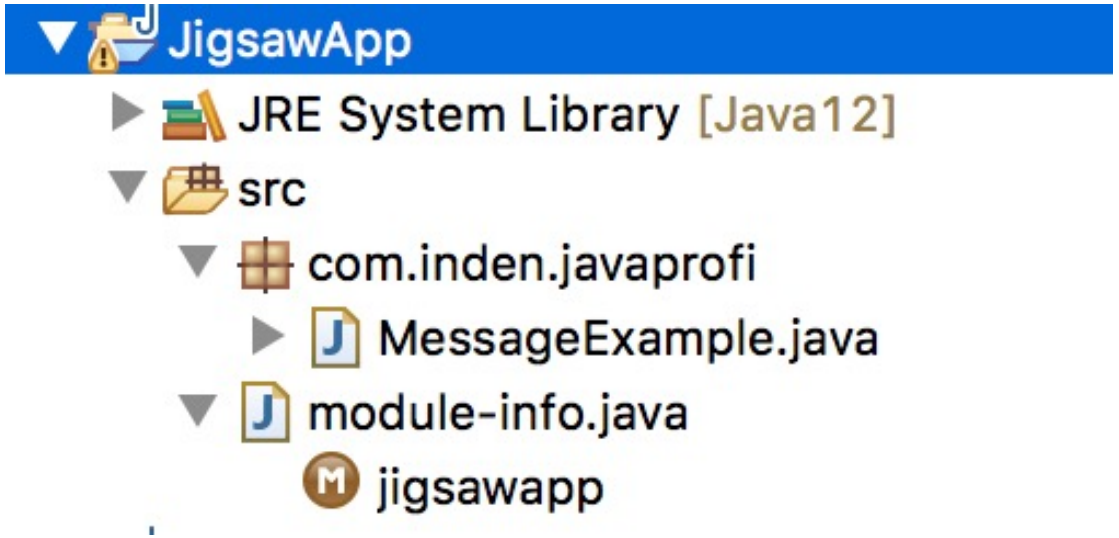
IDE Support

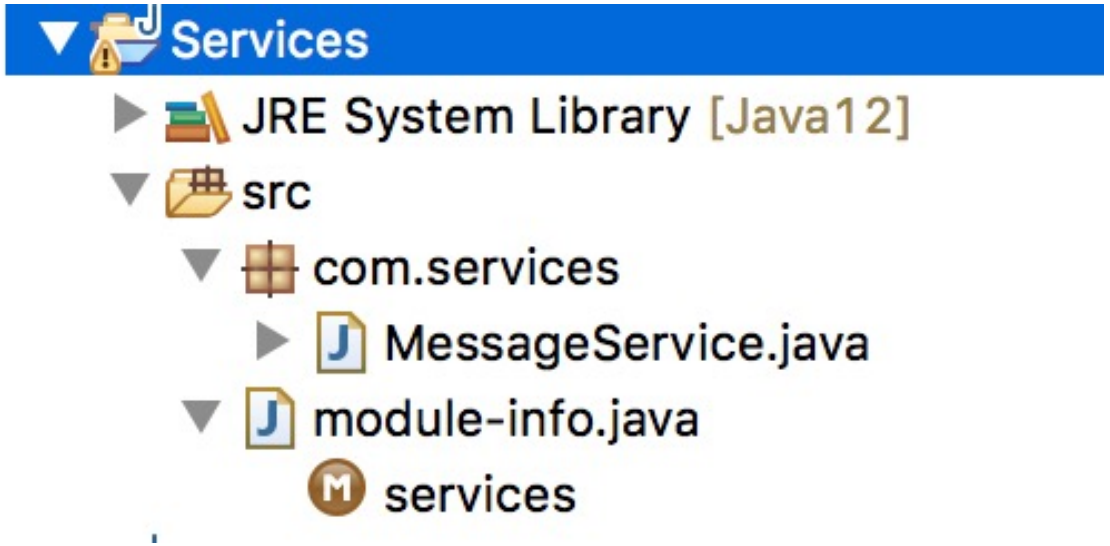
- Current IDEs basically good
- Eclipse: Module correspond to projects
- IntelliJ: Special module construct / concept below projects
- No standard layout yet, various variations possible
- My personal Suggestion: Use the normal layout without the module name in "src".



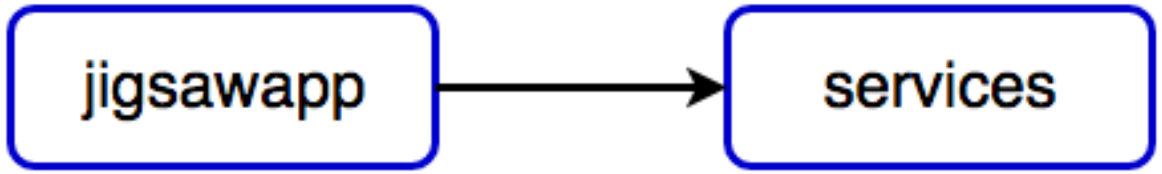
Example 2 Modules: dir layout in Eclipse IDE

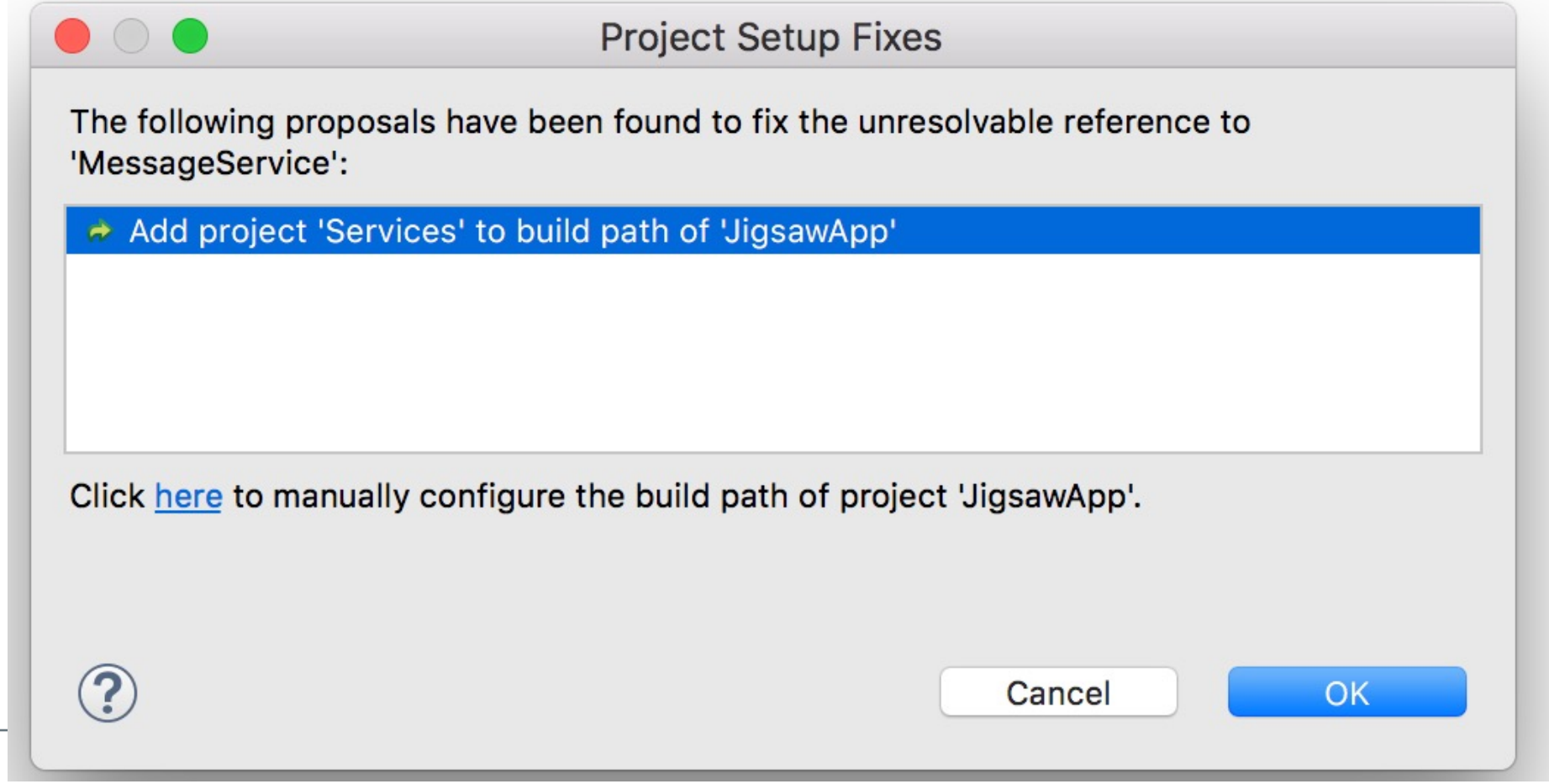






Example 2 Modules: Dependency in Eclipse IDE





Example 2 Modules: Conclusion

- **That's it! Much better than on the console, isn't it?**

- **Interim conclusion:**
  - **Clean structure**
  - **Separate units as projects / modules**
  - **Separate building and further development idenpendently possible**
  - **Easy to bring to the structure expected by build tools**
  - **Better directory structure: Does not require an artificial intermediate directory**
  - **Separate provision as JAR easily possible**

- **BUT: If the directory layout is different to Oracle Layout, the JAR tool still has a bug and we can't build such projects into correct modular JARs! With Java 9 this was still possible**

Specialities

List dependencies

- **Determining of dependencies: `jdeps lib/*.jar`**

```
jigsawapp
 [file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch6/
       ch6_2_6_dependencies_module_example/lib/jigsawapp.jar]
   requires mandated java.base (@9-ea)
   requires services
jigsawapp -> java.base
jigsawapp -> services
   com.inden.javaprofi           -> com.services.api           services
   com.inden.javaprofi           -> java.io                    java.base
   com.inden.javaprofi           -> java.lang                  java.base
   com.inden.javaprofi           -> java.lang.invoke           java.base
services
 [file:///Users/michaeli/Desktop/PureJava9/quelltext/jigsaw_ch6/
       ch6_2_6_dependencies_module_example/lib/services.jar]
   requires mandated java.base (@9-ea)
services -> java.base
   com.services.api              -> com.services.impl          services
   com.services.api              -> java.lang                  java.base
   com.services.impl             -> com.services.api           services
   com.services.impl             -> java.lang                  java.base
```

- **NOTE: When using the IDE we only have to copy the respective JARs into a common lib directory, then we can execute the actions directly!**

List dependencies compactly

- **Compact preparation of dependencies**

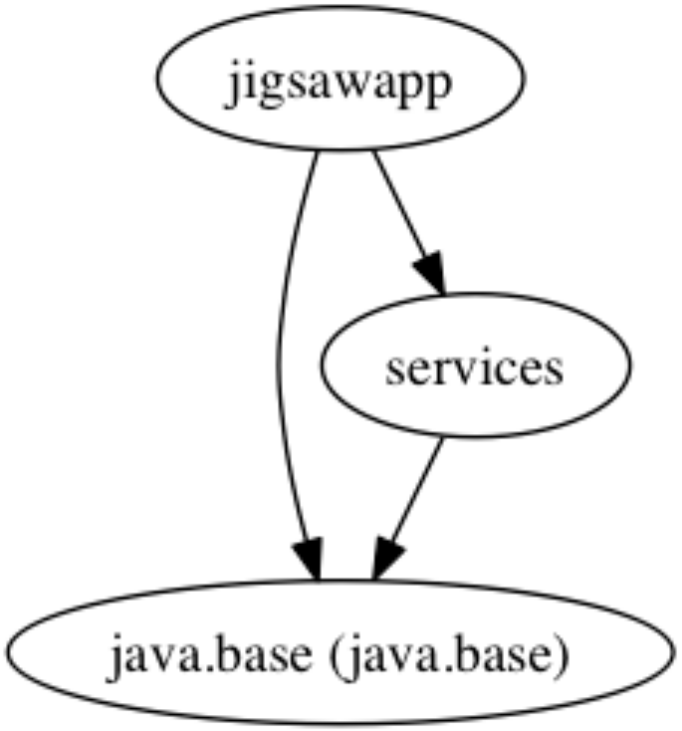
```
jdeps -s lib/*.jar
```

```
jigsawapp -> java.base
jigsawapp -> services
services -> java.base
```

Creation of dependency graph

- **Preparation of a dependency graph**

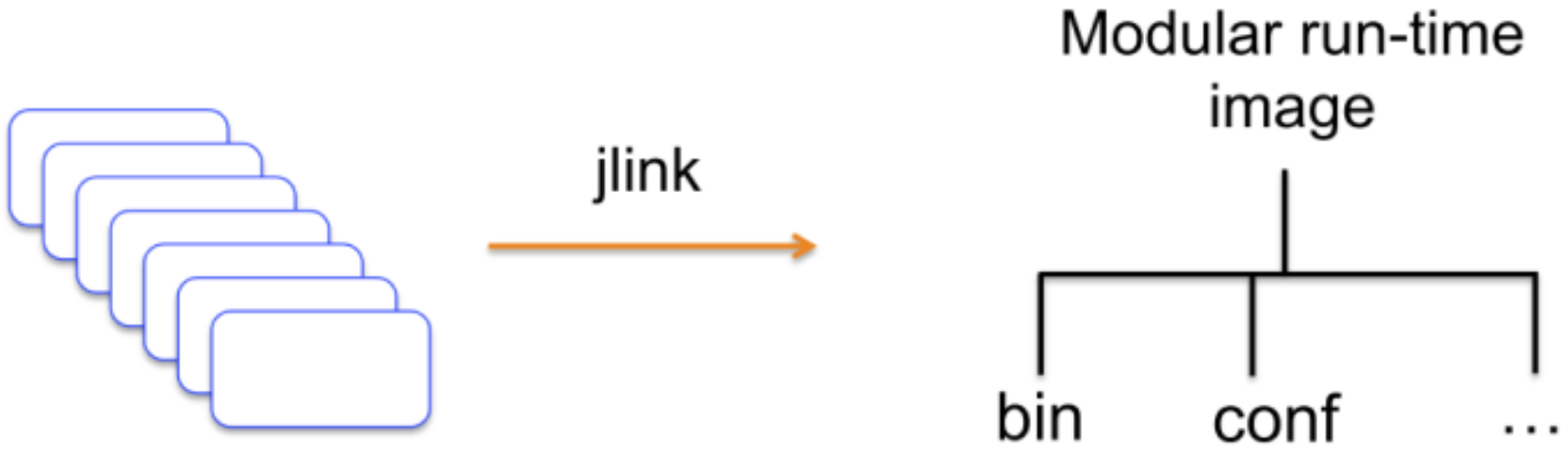
```
jdeps --module-path build -dotoutput graphs lib/*.jar
```



Linking

- **Creation of executable with included Java runtime**

```
jlink --module-path $JAVA_HOME/jmods:lib --add-modules jigsawapp \
      --launcher jigsawapp=jigsawapp/com.inden.javaprofi.MessageExample \
      --output exec_example
```



Linking

```
`-- exec_example
    |-- bin
    |   |-- java
    |   |-- jigsawapp
    |   `-- keytool
    |-- conf
    |   |-- net.properties
    |   `-- security
    |       |-- java.policy
    |       |-- java.security
    |       `-- policy
    |           |-- README.txt
```

- **Starting the application with the following command**
  - `./exec_example/bin/jigsawapp`

Using and Referencing JDK Modules

Using JDK modules

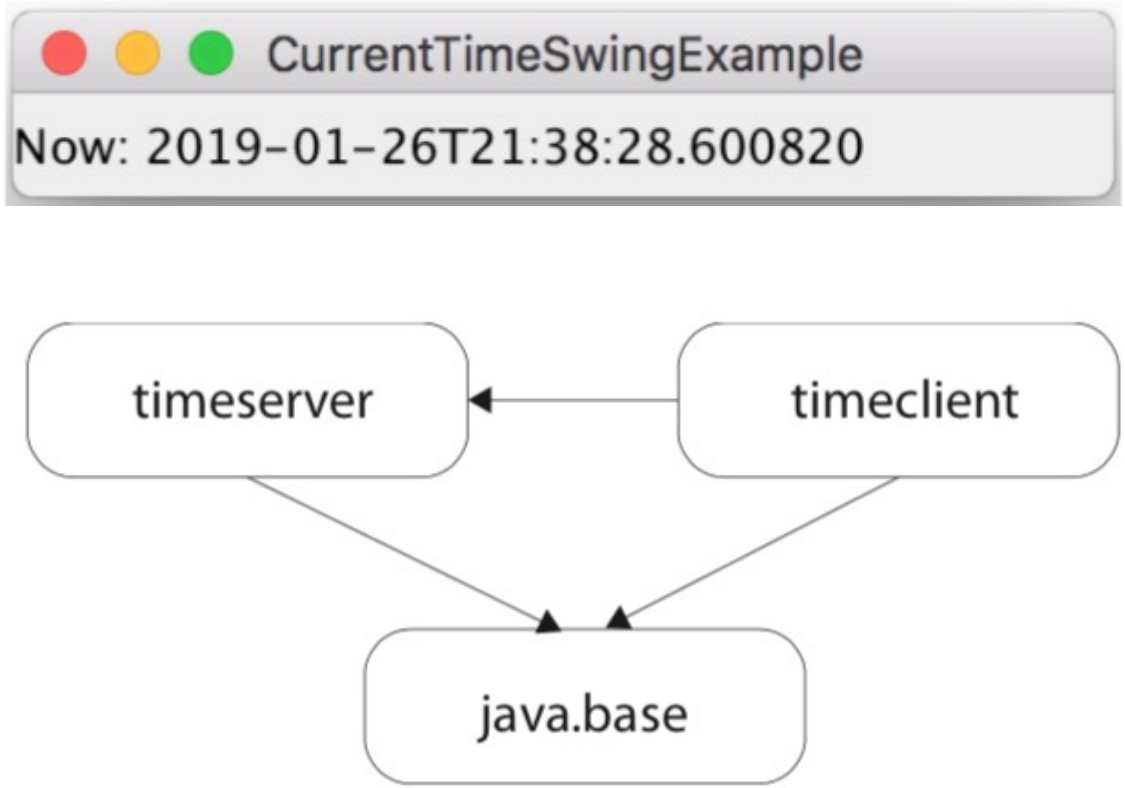
- So far we only used our own functionality => unrealistic
- What if we need things from the JDK?

  => some already in the module `java.base`,
  the base of all modules analog to the class `Object`

  => other modules of the JDK must be explicitly listed in the module descriptor.

Using JDK modules

Example: modularized Swing application, which displays the current time in a window



Using JDK modules

Example: modularized Swing application, which displays the current time in a window

module-info.jav | build.gradle | *HelloJigsaw.ja | Classsresolver. | ReflectionExamp

```
1  package com.hellojigsaw;
2
3  import javax.swing.JFrame;
4  import javax.swing.JLabel;
5
6  public class CurrentTimeSwingExample
7  {
8      public static void main(final String[] args)
9      {
10         final JFrame appFrame = new JFrame("CurrentTimeSwingExample");
11         final JLabel label = new JLabel("Now: " + TimeInfo.getCurrentTime());
12
13         appFrame.getContentPane().add(label);
14
15         appFrame.setSize(300, 50);
16         appFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17         appFrame.show();
18     }
19 }
```

How do we know which modules to include?

Hints by the compiler or the IDE

```
src/timeserver/com/server/TimeInfo.java:4: error: package java.util.logging is
      not visible
import java.util.logging.Level;
                ^
  (package java.util.logging is declared in module java.logg
       timeserver does not read it)
src/timeserver/com/server/TimeInfo.java:5: error: package ja
      not visible
import java.util.logging.Logger;
                ^
  (package java.util.logging is declared in module java.logg
       timeserver does not read it)
src/timeclient/com/client/CurrentTimeSwingExample.java:5: er
      swing is not visible
import javax.swing.JFrame;
            ^
  (package javax.swing is declared in module java.desktop, b
        does not read it)
src/timeclient/com/client/CurrentTimeSwingExample.java:6: er
      swing is not visible
import javax.swing.JLabel;
            ^
  (package javax.swing is declared in module java.desktop, b
        does not read it)
src/timeclient/com/client/CurrentTimeSwingExample.java:24: e
      symbol
        appFraem.pack();
        ^
  symbol:   variable appFraem
  location: class CurrentTimeSwingExample
5 errors
```

```
final JFrame appFrame = new JFrame("CurrentTimeSw
final JLabel label = new JLabel("Now: " + TimeInf
appFra
appFra
appFra
appFra
```

JLabel cannot be resolved to a type

6 quick fixes available:

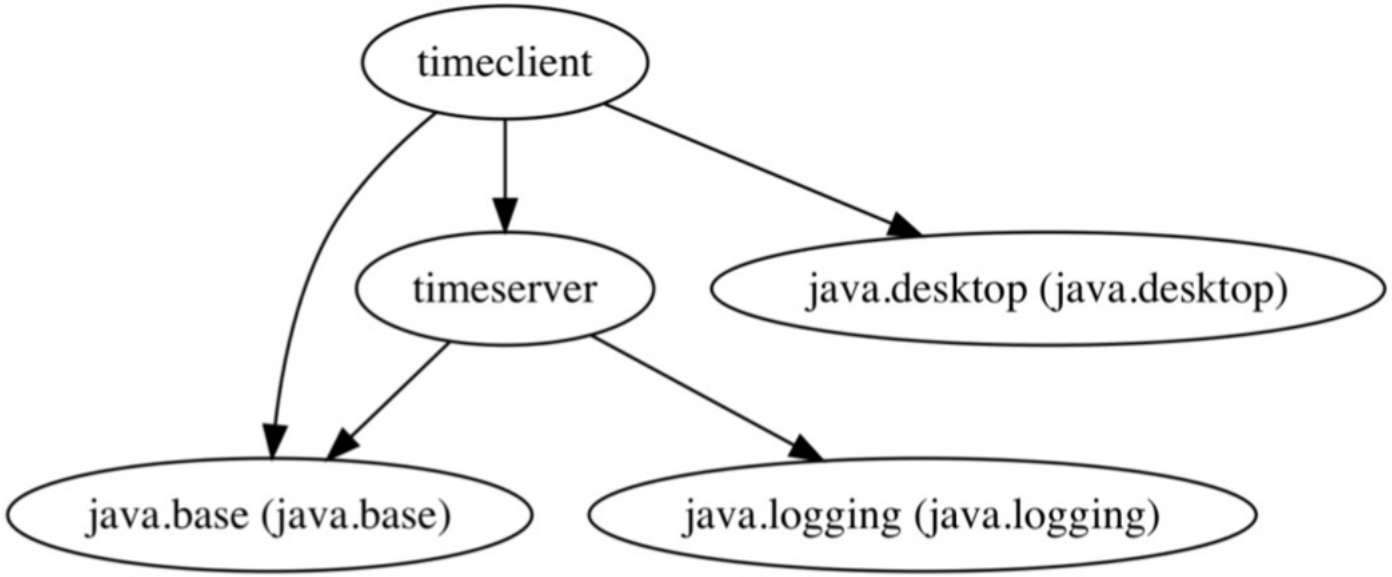
- Create class 'JLabel'
- Add 'requires java.desktop' to module-info.java
- Create interface 'JLabel'
- Create enum 'JLabel'
- Add type parameter 'JLabel' to 'main(String[])'
- Fix project setup...

Press 'F2' for focus

Include modules of the JDK: Adjust module descriptors

```
module timeclient
{
    requires java.desktop;

    requires timeserver;
}
```

```
module timeserver
{
    requires java.logging;

    exports com.server;
}
```



Notes on directory layout

Recap: dir layout (proposed by Oracle)

- Application with several modules => common src directory
- And for every module: source code stored in a subdirectory with the module name

```
`-- src
    |-- com.inden.module1
    |   |-- com
    |   |   `-- inden
    |   |       `-- module1
    |   |           `-- Application.java
    |   `-- module-info.java
    `-- com.inden.module2
        |-- com
        |   `-- inden
        |       `-- module2
        |           `-- OtherClass.java
        `-- module-info.java
```

- Often unsuitable in practice!!!! Why???

What's so problematic about that?

Conclusion for Oracle dir layout

While this directory layout allows compiling in one go (with Multi Module Build), it makes it more difficult to

1. **Achive a separation of modules into different projects (for build tools and IDEs),**
2. **Create JARs as standalone deployables, and**
3. **a clean separation of the modules from each other.**

Consequences:

⇒ The whole thing questions the goals of modularization

⇒ The goal was to create small, self-contained components or modules

⇒ Due to the above arrangement, they become a bit of a monolith again.

So, how it is going better?

Introduction: more appropriate dir layout

Variant 1: Application with several modules => several src directories with module name sub dir

```
playlistservice-java9-modules-example
|-- playlistservice
|   `-- src
|       `-- main
|           `-- java
|               `-- playlistservice
|                   |-- com
|                   |   `-- javaprofi
|                   |       `-- spi
|                   |           `-- PlayListService.java
|                   `-- module-info.java
|-- playlistserviceconsumer
|   `-- src
|       `-- main
|           `-- java
|               `-- playlistserviceconsumer
|                   |-- com
|                   |   `-- serviceconsumer
|                   |       `-- ServiceConsumerExample.java
|                   `-- module-info.java
`-- playlistserviceprovider
    `-- src
```

Introduction: most appropriate dir layout

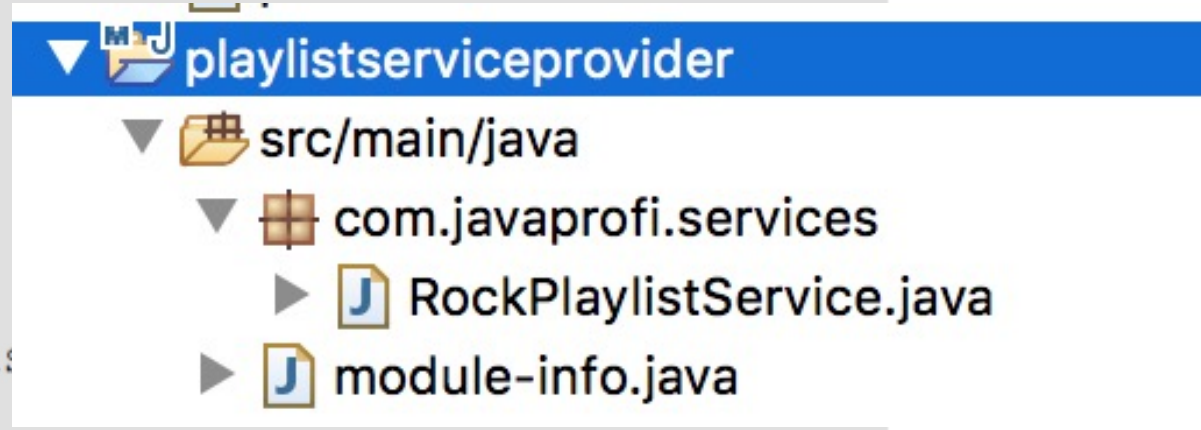
Variant 2: Application with several modules => several normal src directories

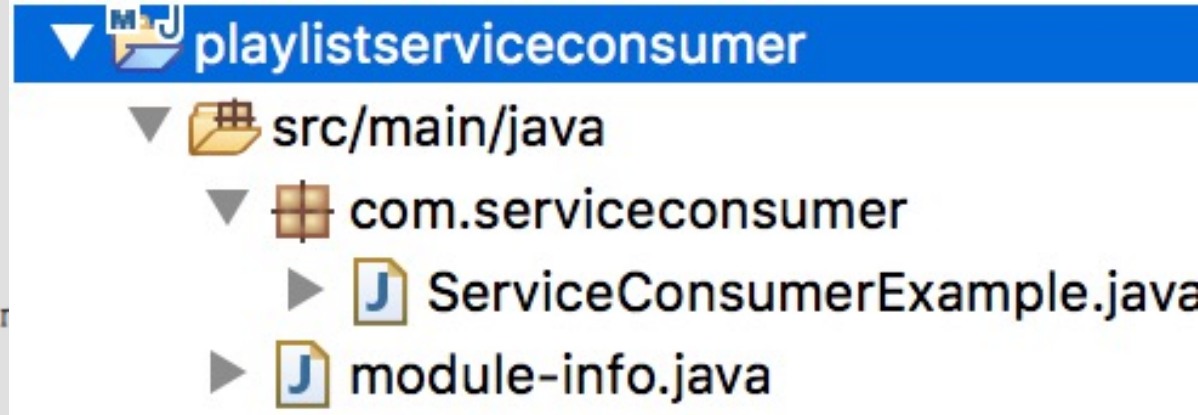
```
playlistservice-java9-modules-example
|-- playlistservice
|   '-- src
|       '-- main
|           '-- java
|               '-- playlistservice
|                   |-- com
|                   |   '-- javaprofi
|                   |       '-- spi
|                   |           '-- PlayListService.java
|                   '-- module-info.java
|-- playlistserviceconsumer
|   '-- src
|       '-- main
|           '-- java
|               '-- playlistserviceconsumer
|                   |-- com
|                   |   '-- serviceconsumer
|                   |       '-- ServiceConsumerExample.java
|                   '-- module-info.java
'-- playlistserviceprovider
    '-- src
```

Introduction: most appropriate dir layout

Variant 2: Application with several modules => several normal src directories

```
playlistservice-java9-modules-example
|-- playlistservice
|   `-- src
|       `-- main
|           `-- java
|               `-- playlistservice
|                   |-- com
|                   |   `-- javaprofi
|                   |       `-- spi
|                   |           `-- PlayList
|                   `-- module-info.java
|-- playlistserviceconsumer
|   `-- src
|       `-- main
|           `-- java
|               `-- playlistserviceconsumer
|                   |-- com
|                   |   `-- serviceconsumer
|                   |       `-- ServiceConsu
|                   `-- module-info.java
`-- playlistserviceprovider
    `-- src
```





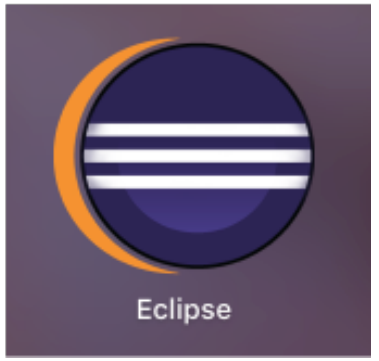
Introduction: dir layout

Example App with 3 modules, here Eclipse variant src and "without" module name

```
.
├── ServiceConsumer
│   └── src
│       ├── com
│       │   └── inden
│       │       └── app
│       │           └── Application.java
│       └── module-info.java
├── ServiceInterface
│   └── src
│       ├── com
│       │   └── inden
│       │       └── interfaces
│       │           └── ServiceIF.java
│       └── module-info.java
└── ServiceProvider
    └── src
        ├── com
        │   └── inden
        │       └── provider
        │           └── MessageService.java
        └── module-info.java
```

IDE & Tool Support

- Current IDEs & Tools basically good



- Eclipse: Modules correspond to projects
- IntelliJ: Special module construction below projects

- No standard layout yet, various variations possible
- Use the normal layout without the module name in "src".
- For modules otherwise configuration of paths necessary ☹

Maven™

- Maven somewhat more comfortable than Gradle due to Multi-Module Build
- Gradle requires manual configuration of the module path

```
tasks.withType(JavaCompile) {
    options.compilerArgs += ["--module-path", classpath.asPath]
}
```

PART 2
Visibilities and transitive dependencies

Visibilities

Visibilities / Access control

Visibilities in JDK 8

- `private` – only visible in own class
- Default – visible in package
- `protected` – as default, but also visible for derived class
- `public` – visible for all classes

=> PUBLIC means visible for EVERYONE in the CLASSPATH!!

=> NO Access Control

Visibilities / Access control

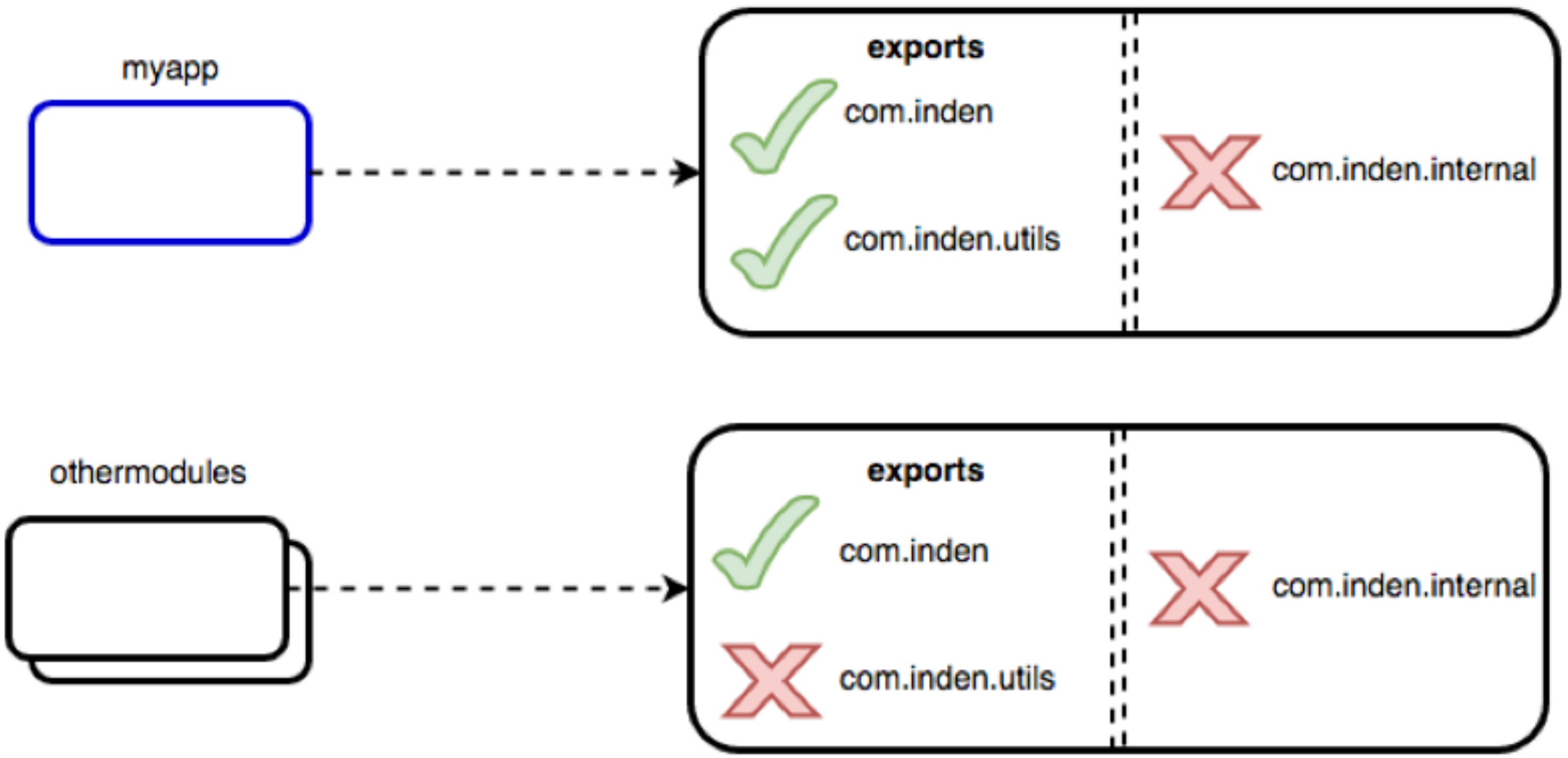
Visibilities in JDK 9

With the introduction of modularization, the visibility of types can be specified more precisely. As a rule, this is only relevant for types defined as public.

- **Exported Globally (`exports + requires`) – Public for all modules who are reading (requires)**
- **Qualified export (`exports to + requires`) – just visible for specified modules and those who are reading**
- **Module intern (no export) – just visible in the module itself**

Visibility control

```
module specialexport
{
    exports com.inden.utils to myapp;
    exports com.inden;
}
```



Visibility control

Qualified Export (export … to …)

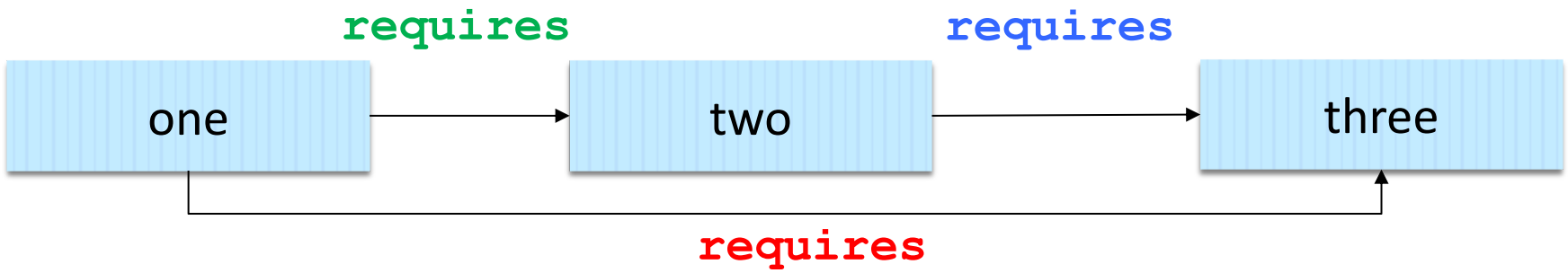
- access not specified / permitted in the module descriptors are prevented.
- a restricted export for a defined number of modules is possible.

```
module java.base
{
     exports sun.reflect to java.corba,
                            java.logging,
                            java.sql,
                            java.sql.rowset,
                            jdk.scripting.nashorn;
}
```

Transitive dependencies (Implied Readability)

Transitive Dependencies

- **The dependencies described in module descriptors are not automatically propagated to using modules.**
- **Whenever several modules are combined, this can become cumbersome:**



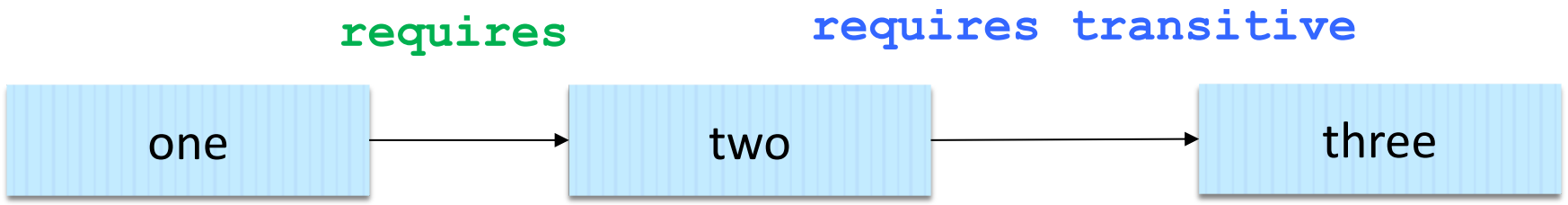

```
public class A {
  public void bar() {
    B b = new B();
    b.foo();  // ok

    // requires three
    C c = b.foo();
  }
}
```

```
public class B {
  public C foo() {
    ...
  }
}
```

```
public class C
{}
```

Transitive Dependencies

transitive dependency can be explicitly declared as such




```
public class A {
  public void bar() {
    B b = new B();
    b.foo();  // ok

    // requires three
    C c = b.foo();
  }
}
```

```
public class B {
  public C foo() {
    ...
  }
}
```

```
public class C
{}
```

Transitive Dependencies

Readability in the Java SE module graph

```
module java.sql {
  requires java.logging;
  exports java.sql;
}
```

```
module java.logging {
  exports java.util.logging;
}
```

```
package java.sql;
import java.util.logging.Logger;
public interface Driver {
  Logger getParentLogger();
}
```

```
package java.util.logging;
public class Logger {
  ...
}
```

http://openjdk.java.net/projects/jigsaw/talks/jigsaw-under-the-hood-j1-2015.pdf

Transitive Dependencies

Readability in the Java SE module graph

```
module myApp {
  requires java.sql;
  requires java.logging; ☹
}
```

```
module java.sql {
  requires java.logging;
  exports java.sql;
}
```

```
module java.logging {
  exports java.util.logging;
}
```

http://openjdk.java.net/projects/jigsaw/talks/jigsaw-under-the-hood-j1-2015.pdf

Transitive Dependencies

Readability in the Java SE module graph

```
module myApp {
  requires java.sql;
  requires java.logging; ☺
}
```

```
module java.sql {
  requires transitive java.logging;
  exports java.sql;
}
```

```
module java.logging {
  exports java.util.logging;
}
```

http://openjdk.java.net/projects/jigsaw/talks/jigsaw-under-the-hood-j1-2015.pdf

Transitive Dependencies

Implied Readability (Aggregator Module)

- It is often practical to bundle different modules into larger units. This is possible with `requires transitive`.

```
java.se@9-ea
  requires mandated java.base
  requires transitive java.compiler
  requires transitive java.datatransfer
  requires transitive java.desktop
  requires transitive java.instrument
  requires transitive java.logging
  requires transitive java.management
  requires transitive java.management.rmi
  requires transitive java.naming
  requires transitive java.prefs
  requires transitive java.rmi
  requires transitive java.scripting
  requires transitive java.security.jgss
  requires transitive java.security.sasl
  requires transitive java.sql
  requires transitive java.sql.rowset
  requires transitive java.xml
  requires transitive java.xml.crypto
```

PART 3
Handle dependencies with services

Dependencies between Modules

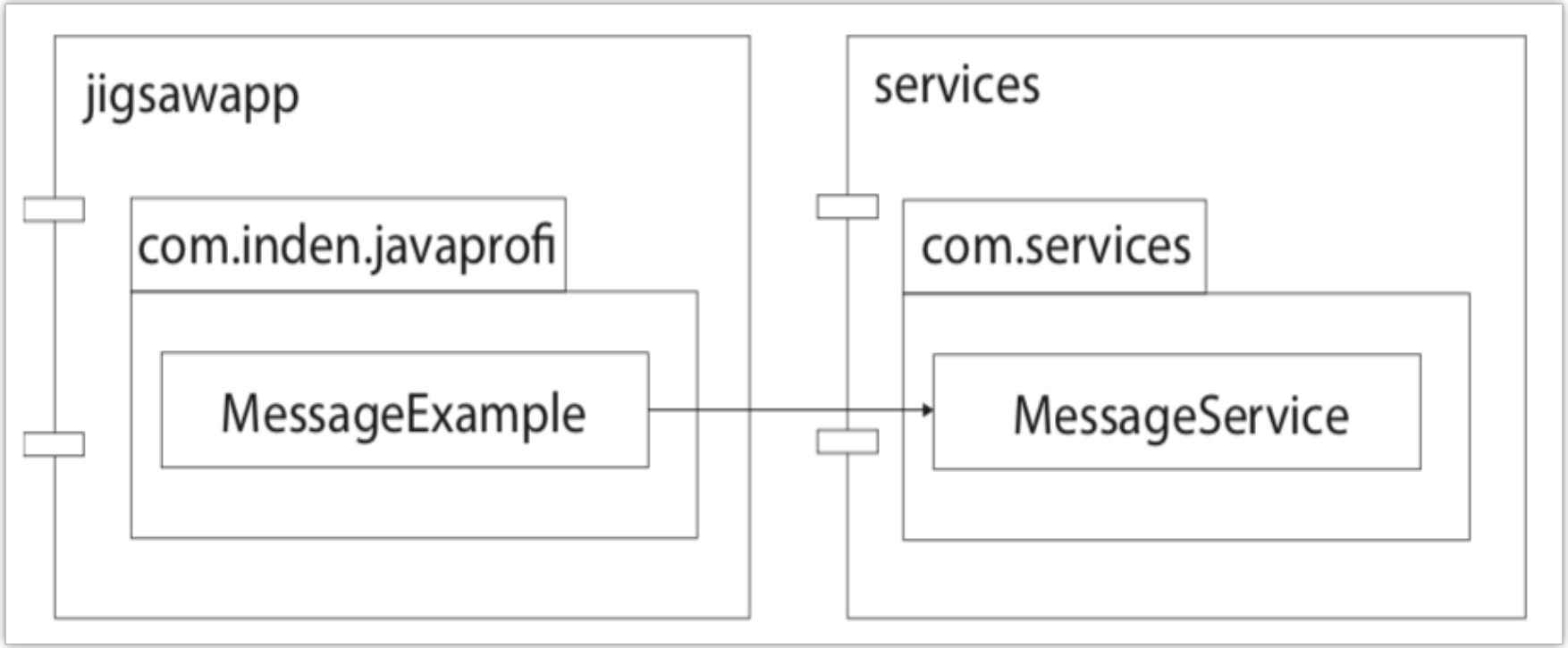
Modules allow an application to be subdivided into several components:

- **But so far: A module refers directly to another or more precisely a class uses a class of another module directly.**
- **This is not really problematic for the use of JDK classes**
- **But for own applications this is sometimes questionable => stronger implementation dependency**

What's so problematic about that?

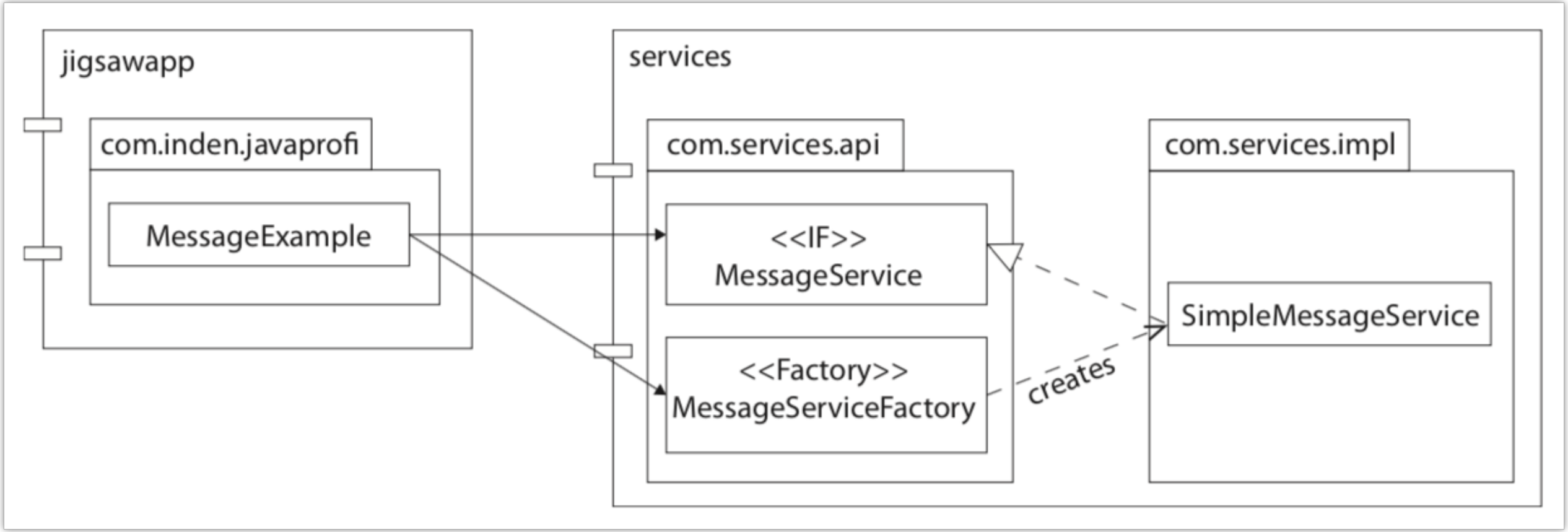
Tight coupling

If there is a direct dependency between modules, they are tightly coupled and logically form one module.



Achieve decoupling (with Factory)

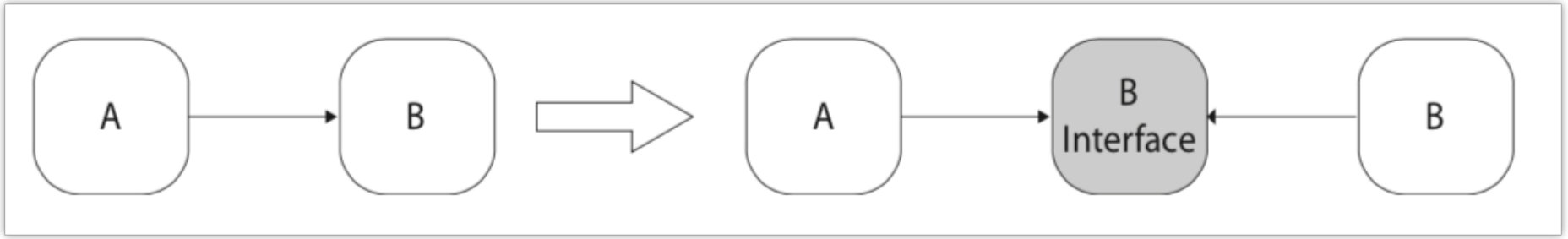
If a stronger decoupling is desired, a using application ideally only refers to interfaces and obtains the realizations via factory methods or services.



Achieve decoupling (with Factory)

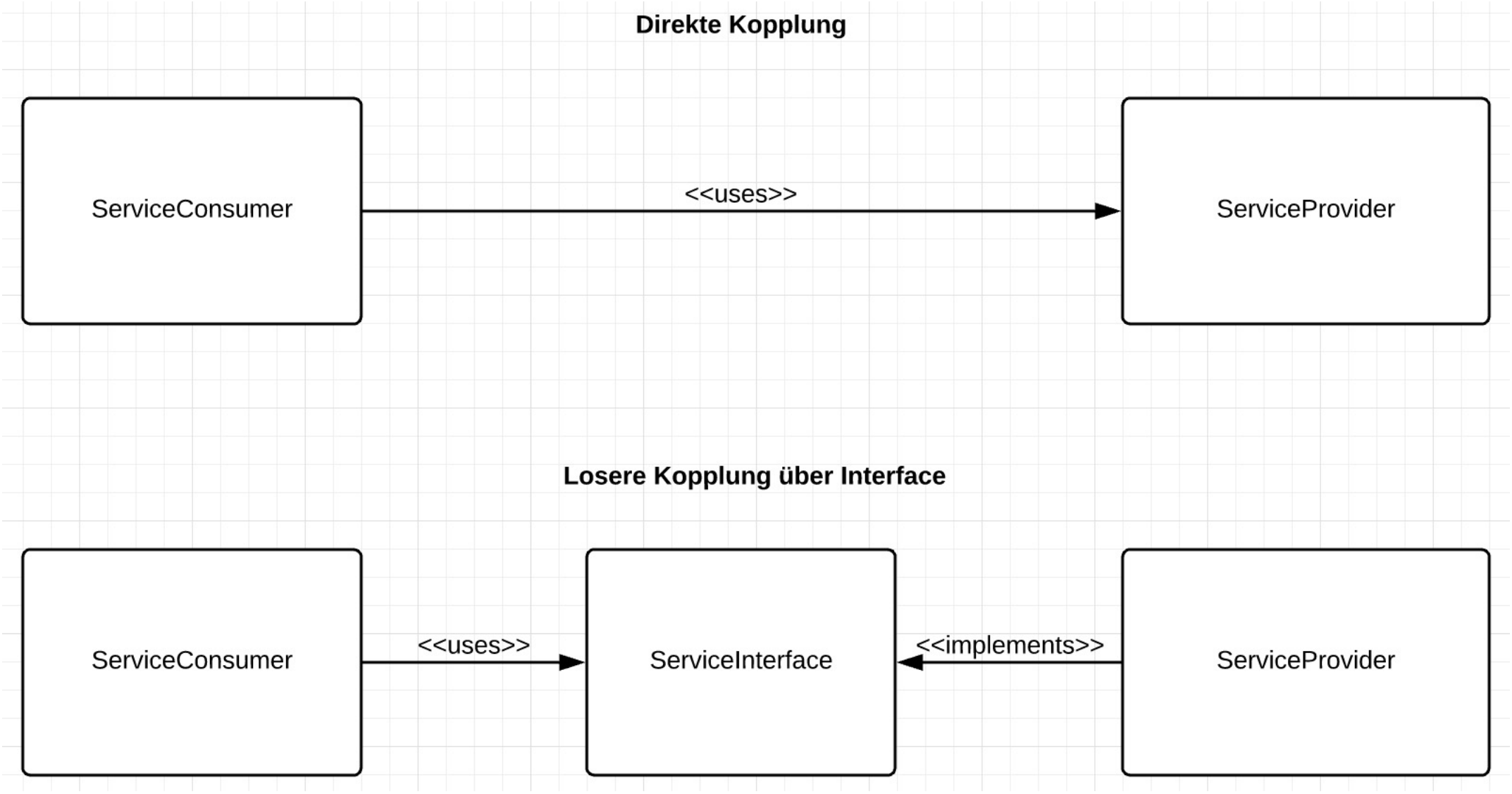
What do we achieve through the introduction of Indirection and Factory?

1. **We are still functionally on the same level as before, but no (strong) implementation dependency on concrete classes anymore - only the dependency on the factory class and the interface remains.**
2. **A more advanced variant => additional module with interface definitions to which both modules refer (Dependecy Inversion Principle).**



3. **Services are a way of how to further reduce the coupling.**

Services – Consumer, Interface and Provider



Services – Since Java 6

Services require 3 steps:

1. Definition of a **Service Interface**
2. Definition of a **Service Provider**
3. Definition of a **Service Consumer** using the ServiceLoader class

Services – With Java 6

- **Class java.util.ServiceLoader**
  - enables loose coupling
  - allows to include various implementations of an interface or an abstract class as a service at runtime.
  - Implementations are loaded at runtime and provided as iterators<T>:

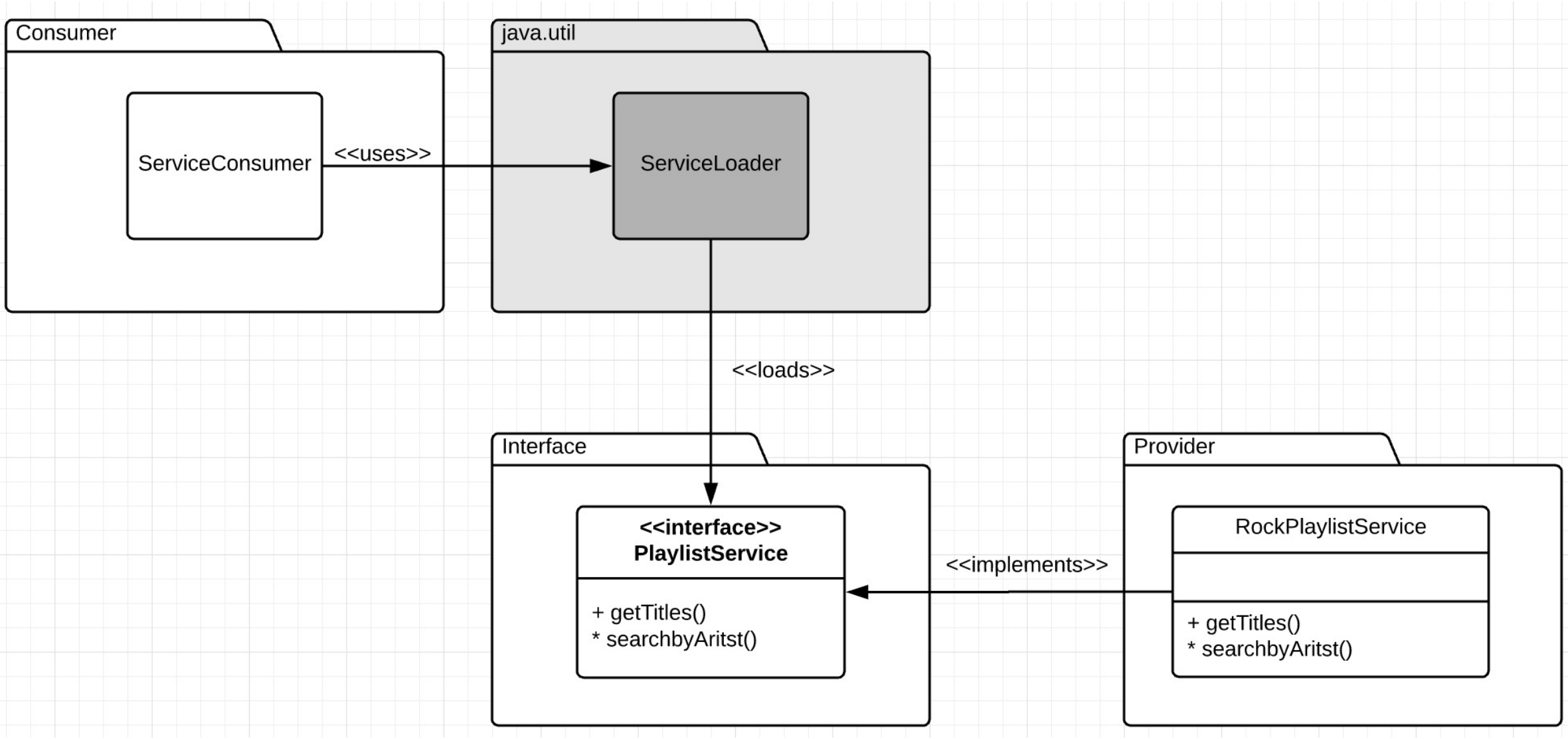
```
final Iterator<DesiredServiceInterface> iterator =
ServiceLoader.load(DesiredServiceInterface.class).iterator();
```

  - The ServiceLoader searches the CLASSPATH for this. There are special mechanisms for managing modules in the module descriptors.

DEMO

ServiceLoaderExample

Services – The Application which we want to implement



Services – Step 1: Service Interface

Services require 3 steps:

1. Definition of a Service Interface

```
public interface PlaylistService
{
    public List<String> getTitles();
    public List<String> searchByArtist(final String artist);
}
```

2. Definition of a Service Provider

3. Definition of a ServiceConsumer

Services – Step 2: Definition of a Service Provider

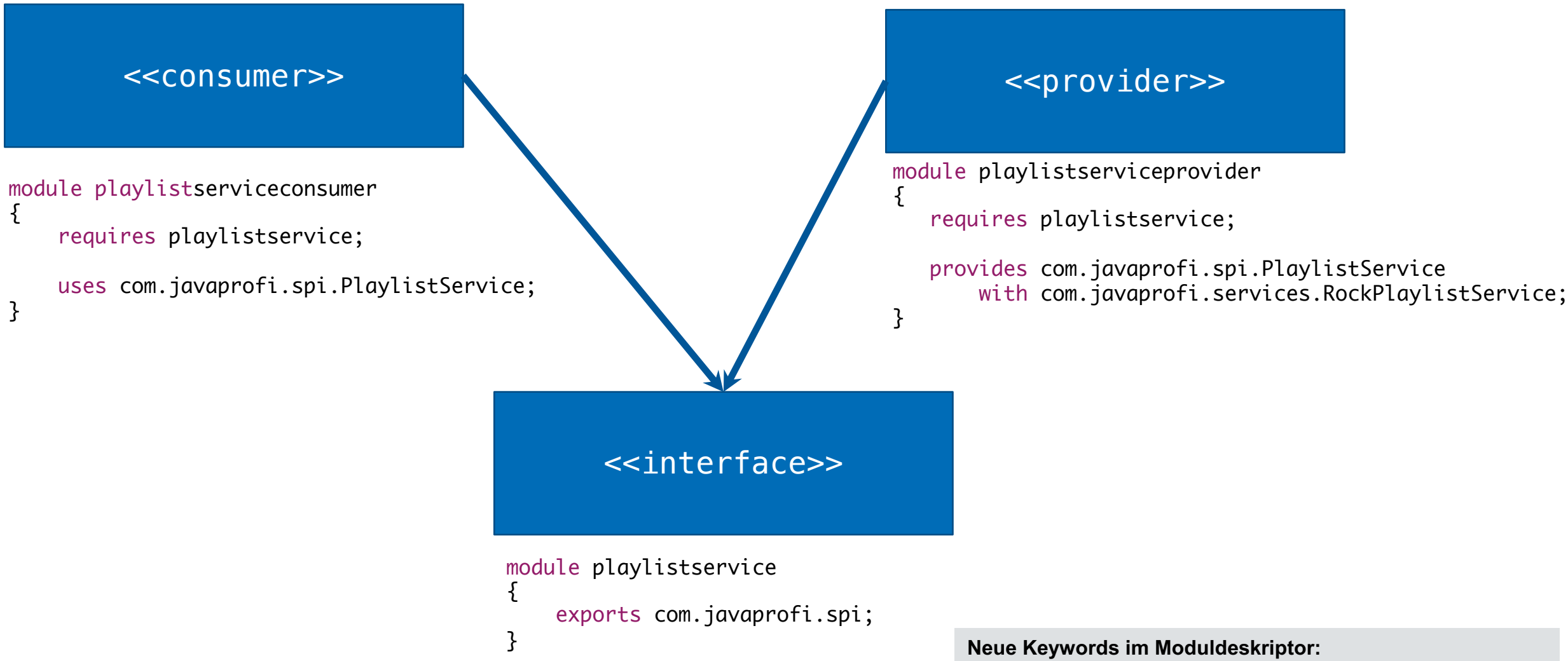
```
public class RockPlaylistService implements PlaylistService
{
    private final Map<String, List<String>> songMap = Map.of("Bryan Adams", List.of("Summer of '69"),
                  "Bon Jovi", List.of("Livin' On A Prayer"), "Metallica", List.of("Nothing Else Matters"),
                  "Nickelback", List.of("How You Remind Me"), "Toto", List.of("Africa", "Hold The Line"));


    @Override
    public List<String> getTitles()
    {
        final Stream<List<String>> titlesStream = songMap.values().stream();
        return titlesStream.reduce(new ArrayList<>(), (a, b) -> {a.addAll(b); return a;});
    }


    @Override
    public List<String> searchByArtist(final String artist)
    {
        return songMap.getOrDefault(artist, List.of("No title found for " + artist));
    }
}
```

- must be **public** and have a **No-Arg constructor** so that the ServiceLoader can load the class and instantiate it by **reflection**.

Services – Step 3: ServiceConsumer & Access via ServiceLoader

```
public static void main(final String[] args) throws Exception
{
    final Optional<PlaylistService> optService = lookup(PlaylistService.class);

    optService.ifPresentOrElse(service -> useService(service),
                               () -> System.err.println("No service provider found!"));
}

private static void useService(final PlaylistService service)
{
    System.out.println(service.getClass());

    final List<String> allTitles = service.getTitles();
    System.out.println("All titles: " + allTitles);
}

private static <T> Optional<T> lookup(final Class<T> clazz)
{
    final Iterator<T> iterator = ServiceLoader.load(clazz).iterator();
    return iterator.hasNext() ? Optional.of(iterator.next()) : Optional.empty();
}
```

Services – Using Modularization / target architecture




```
module playlistserviceconsumer
{
    requires playlistservice;

    uses com.javaprofi.spi.PlaylistService;
}
```

```
module playlistserviceprovider
{
    requires playlistservice;

    provides com.javaprofi.spi.PlaylistService
        with com.javaprofi.services.RockPlaylistService;
}
```

```
module playlistservice
{
    exports com.javaprofi.spi;
}
```

Neue Keywords im Moduldeskriptor:

- `provides <Service-IF-Name> with <ClassName>`
- `uses <Service-IF-Name>`

Services – Step 1: Definition of a Service Interface

a) Definition of a Service Interface (analogous to before)

```
package com.javaprofi.spi;

import java.util.List;

public interface PlaylistService
{
    public List<String> getTitles();
    public List<String> searchByArtist(final String artist);
}
```

b) Definition of a module descriptor

```
module playlistservice
{
    exports com.javaprofi.spi;
}
```

Services – Step 2: Definition of a Service Provider

a) Implementation of the Service Provider (analogous to before)

```
package com.javaprofi.services;
…

import com.javaprofi.spi.PlaylistService;

public class RockPlaylistService implements PlaylistService
{ … }
```

a) Definition of a module desxriptor

```
module playlistserviceprovider
{
   requires playlistservice;

   provides com.javaprofi.spi.PlaylistService
       with com.javaprofi.services.RockPlaylistService;
}
```

Services – Step 3: Service Consumer using the ServiceLoader

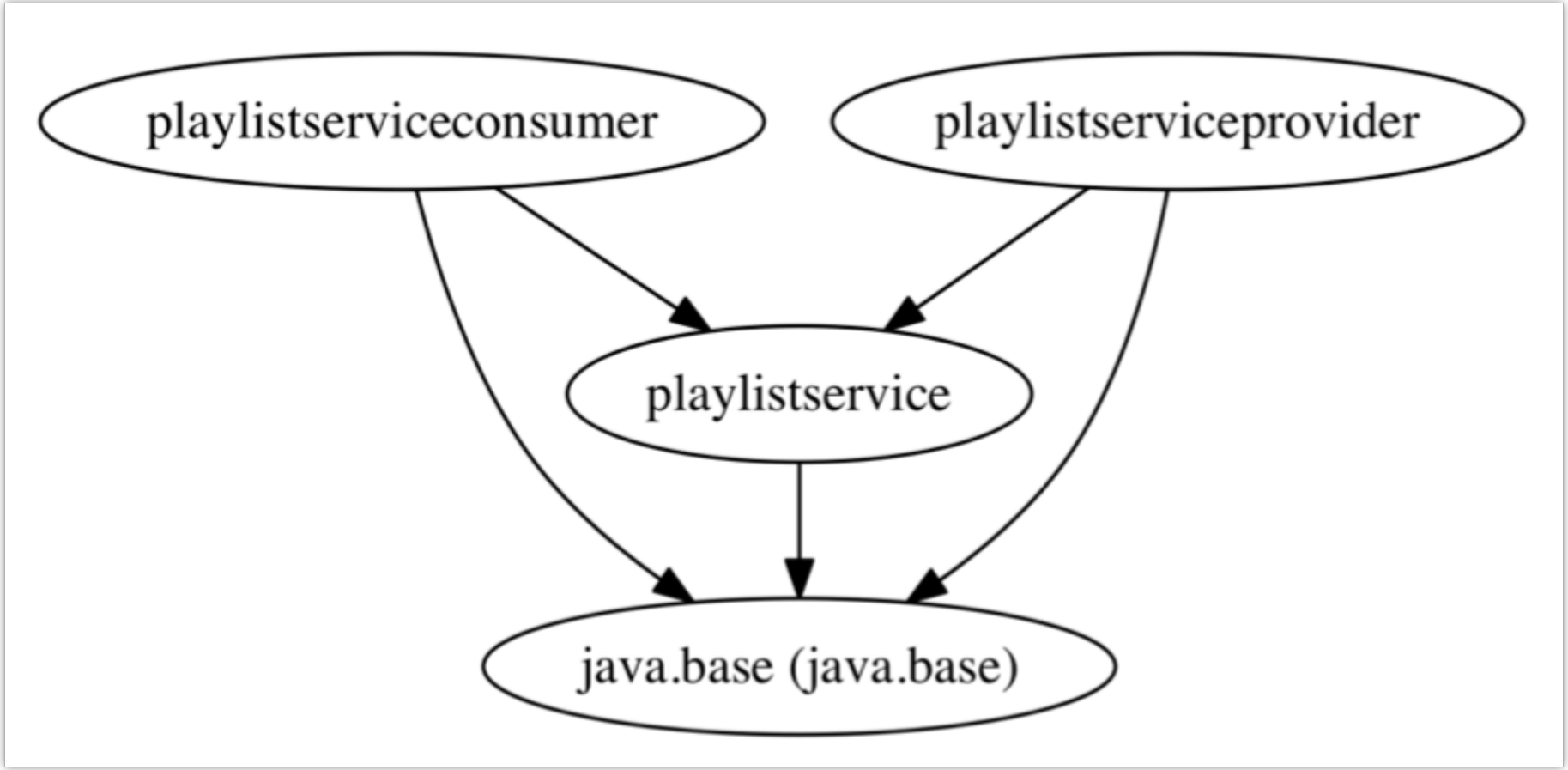
a) Implementation of the Service Consumers (analogous to before)

```
package com.serviceconsumer;
…

import com.javaprofi.spi.PlaylistService;

public class ServiceConsumer
{ … }
```

b) Definition of a module descriptor

```
module playlistserviceconsumer
{
    requires playlistservice;

    uses com.javaprofi.spi.PlaylistService;
}
```

Services – Check of resulting dependencies



Demo PlaylistService

- playlistservice-java11-modules-example
  - commonlibs
  - commonlibsMVN
  - playlistservice
    - build
    - src
      - main
        - java
          - com
            - javaprofi
              - spi
                - PlaylistService.java
              - module-info.java
    - target
    - build.gradle
    - pom.xml
  - playlistserviceconsumer
  - playlistserviceprovider
  - build.gradle
  - pom.xml
  - settings.gradle
- playlistserviceconsumer

PART 4: Including External Modules / Migrations

How do I use other JARs?

3rd Party JARs

Inclusion of 3rd Party Libs / JARs

- **Many libraries are not yet designed for Jigsaw! What now?**

  1. **Compatibility mode** all from CLASSPATH
  2. Migration with **Automatic Modules**

- Automatic Modules emerge when a conventional JAR is included in the Module Path. Example guava-22.0.jar

```
module mymodule
{
    requires guava;
}
```

Types of Modules

Types of Modules I

- **Named Platform Modules -** As is well known, the JDK was subdivided into modules as part of Project Jigsaw. These special modules of the JDK do not have access to the module path.

- **Named Application Modules -** are modules that bundle applications or libraries. JARs that contain a module descriptor have. These modules have access to module path, but not to classes from the CLASSPATH.

- **Open Modules -** Like the first two types of modules, however, all packages are accesible via reflection to the outside world.

Types of Modules II

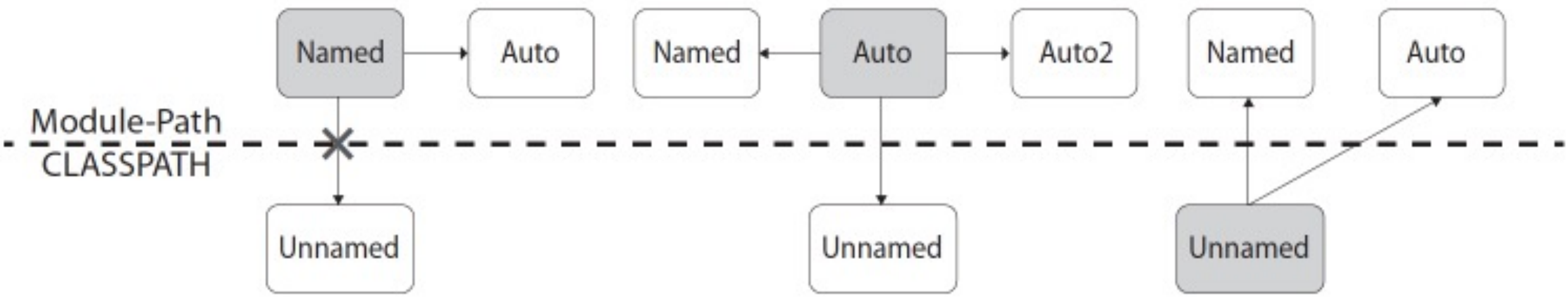
- **Automatic Modules** – Ordinary JAR files, i.e. without module-info.class. The JAR file name without version number and extension is used as module name. ***Automatic Modules export all their packages and can access all modules from the module path as well as JARs from the CLASSPATH.***

- **Unnamed Modules** – In addition to the module path, you can specify a CLASSPATH when compiling or starting programs. All types available there are combined to the so-called unnamed module.

Types of modules III

Modultyp	Origin	Exports	Readability
Platform	JDK	Via exports	-/-
Application	Modular JAR	Via seports	Platform, Application, Automatic
Automatic	JAR without module descriptor	all packages	Platform, Application, Automatic, Unnamed
Unnamed	JAR from CLASSPATH	all packages	Platform, Application, Automaticd, Unnamed

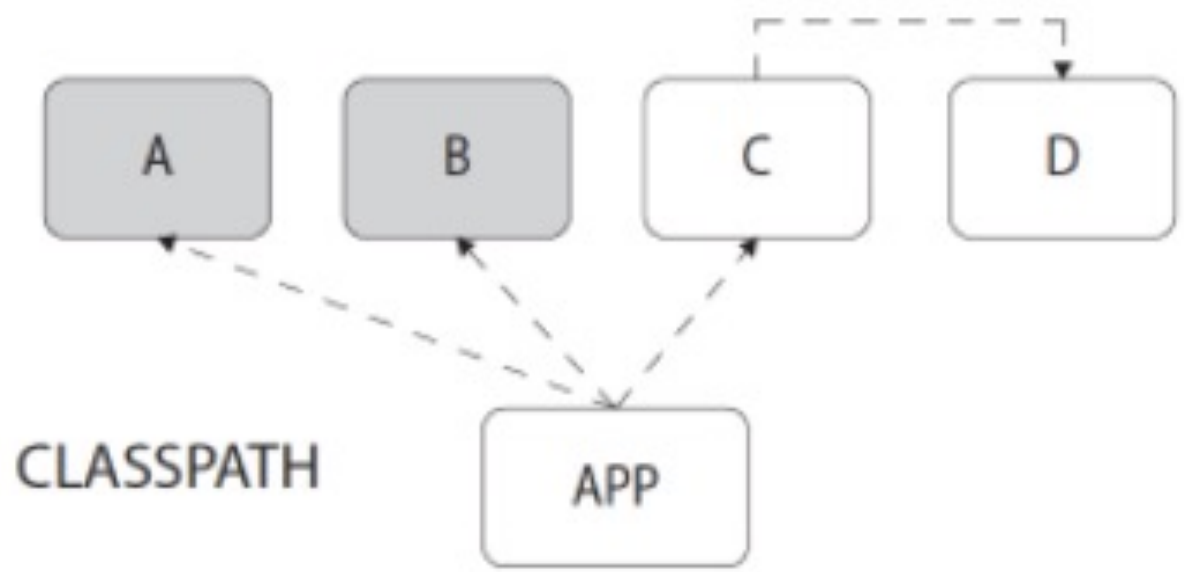
Access options for various types of modules



Migrations

Migration: 3rd Party

Let's consider a migration of an application consisting of several JARs in CLASSPATH, such as app.jar , A.jar , B.jar and C.jar:



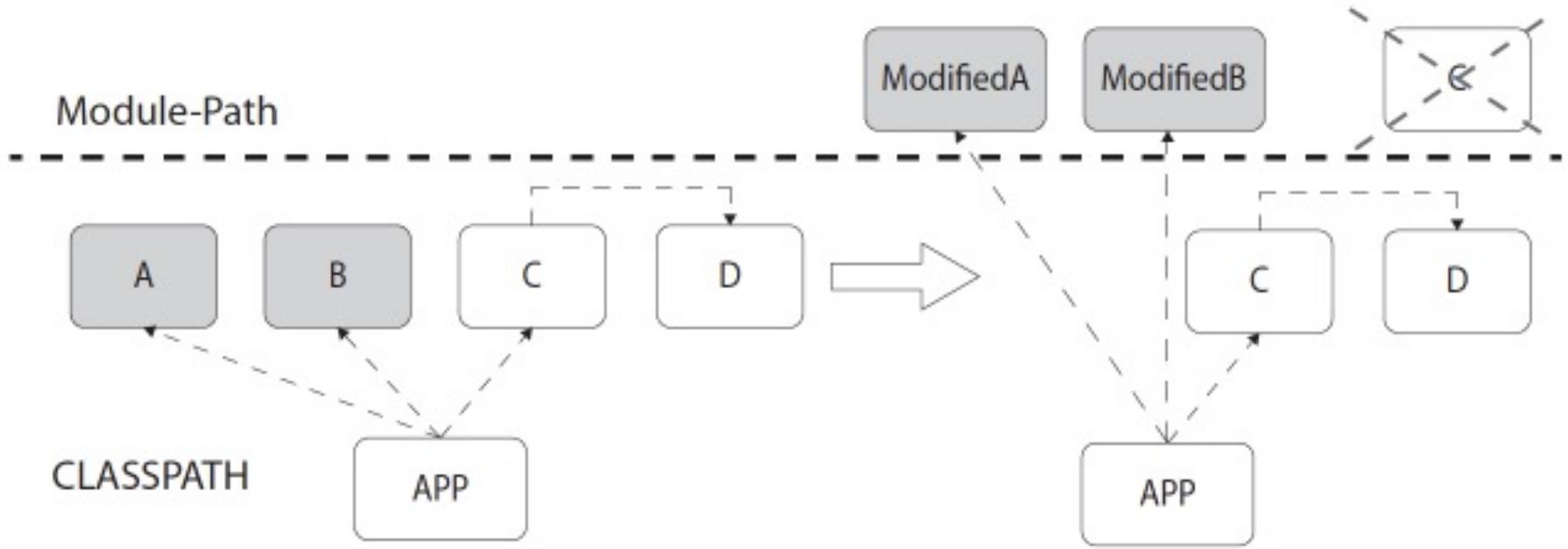
3rd Party Libs

In a so-called bottom-up migration, these JARs are incrementally converted into modular JARs. The following two scenarios can be identified:

- **Bottom-up migration and Named Modules** - If we have the source code of a JAR accessible, we can convert a JAR into a modular JAR by adding a suitable module descriptor and recompiling and re-packaging the module.

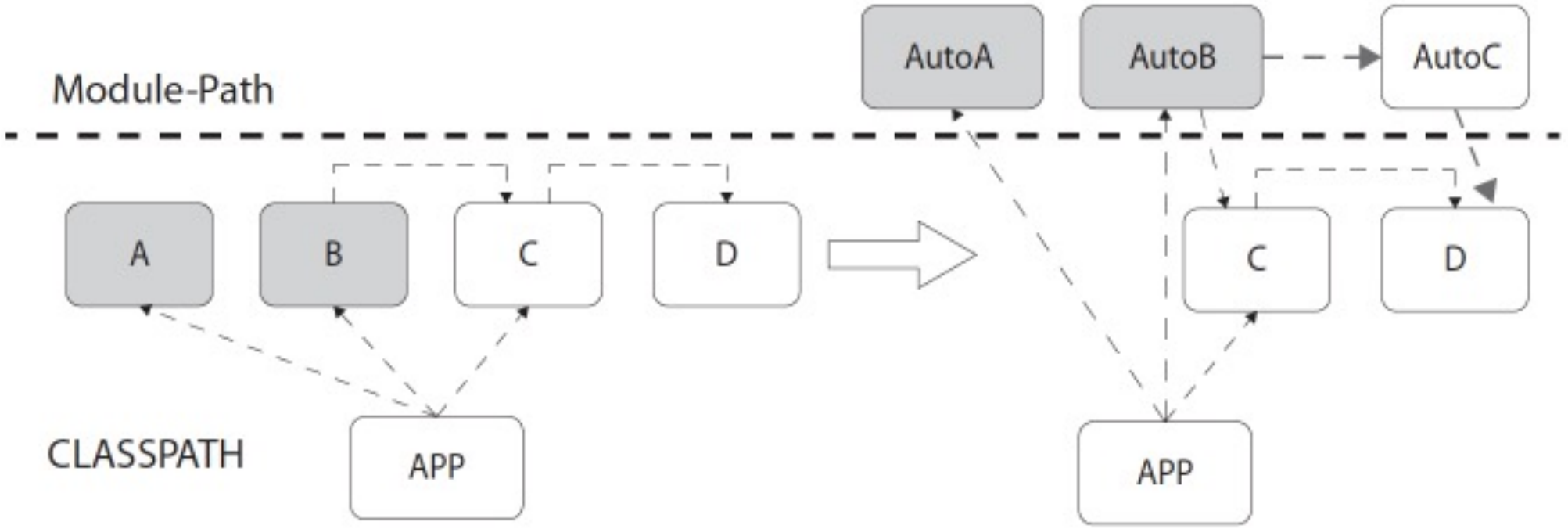
- **Bottom-up Migration and Automatic Modules** - If there is no access to the JAR source code, the JAR cannot be converted into a modular JAR in the manner described above. A good alternative is to use the JAR as an automatic module, possibly in combination with the unnamed module.

Bottom-up-Migration and Named Modules



We can only convert a JAR into a modular JAR if it has no dependencies on other, non-modular JARs of CLASSPATH, because these cannot be referenced in the module descriptor.

Bottom-up-Migration and Automatic Modules



Tips on how to migrate

1. Start with JARs without dependencies
2. JARs with dependencies – use automatics modulles
3. Own Application – this is the last step

Remember: Not every application can be modularized reasonably!
Keep an eye on business value!

Questions?

Help


HELP
IS ON THE
WAY


Java Challenge
Michael Inden
Fit für das Job-Interview und die Praxis –
mit mehr als 100 Aufgaben
und Musterlösungen
dpunkt.verlag


Michael Inden
Java
Die Neuerungen in
Version 9 bis 14
Modularisierung, Syntax- und
API-Erweiterungen
dpunkt.verlag


5.
Auflage
Michael Inden
Der Weg zum
Java-Profi
Konzepte und Techniken für die
professionelle Java-Entwicklung
dpunkt.verlag

Thank You