

Software-Engineering 2

Prof. Dr.-Ing. Gerhard Wanner

Email: wanner@hft-stuttgart.de

LARGE-SCALE SOFTWARE SYSTEMS WITH JAVA

Overview

- Motivation
- Possible solutions
- OSGi
- Java Platform Module System (JPMS).

Motivation – Service Orientation and Complexity

- Most Java applications are manageable enough in isolation
- Problems arise when several (simple) applications work together
 - Applications loaded within the same VM/classpath
 - class/versioning conflicts
 - It is difficult to find out if needed parts are already loaded
 - duplicate or unnecessary parts
 - Some Java application deployments require a hefty 1 or 2 GB of memory to run properly in enterprise settings
 - large memory requirements.

A simple Example – 1

```
package de.hftstuttgart.services;  
  
public class Service {  
    public String call(){  
        return "This is Service Version 1";  
    }  
}
```

This resides in
the Eclipse-
Project
"Service v1"

```
package de.hftstuttgart.services;  
  
public class Service {  
    public String call(){  
        return "This is Service Version 2";  
    }  
}
```

This resides in
the Eclipse-
Project
"Service v2"

A simple Example – 2

```
package de.hftstuttgart.provider;

import de.hftstuttgart.services.Service;

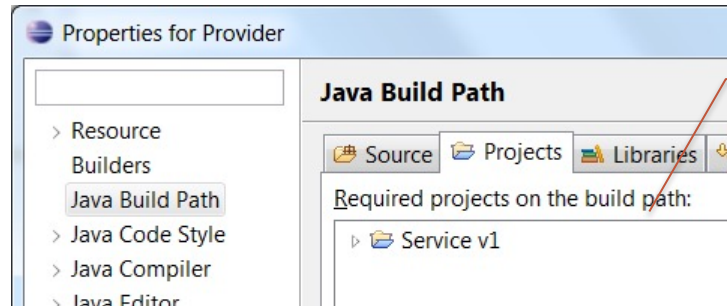
public class ServiceProvider {
    private Service service = new Service();

    public String provide(){
        return service.call();
    }
    public static void main(String[] args){
        ServiceProvider provider = new ServiceProvider();
        System.out.println("Provider:\t" + provider.provide());
    }
}
```

This resides in
the Eclipse-
Project
"Provider"

A simple Example – 3

- Provider should run with Version 1



"Service v1" is
put on the build
path of
"Provider"

- Output:
Provider: This is Service Version 1

A simple Example – 4

```
package de.hftstuttgart.client;

import de.hftstuttgart.provider.ServiceProvider;
import de.hftstuttgart.services.Service;

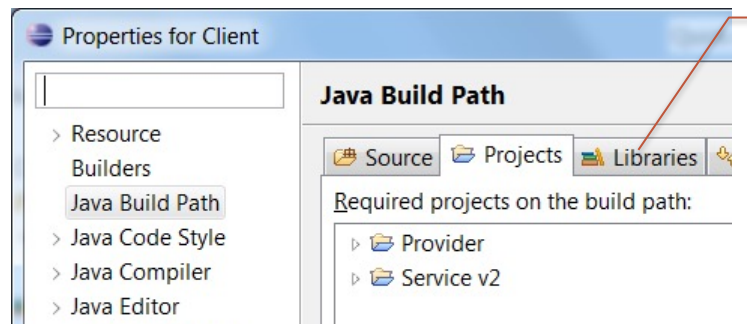
public class Client {
    private Service service = new Service();
    private ServiceProvider provider = new ServiceProvider();

    public String getService(){
        String result = "Calling Service:\t" + service.call() +
            "\nCalling Provider:\t" + provider.provide();
        return result;
    }
    public static void main(String[] args){
        Client client = new Client();
        System.out.println(client.getService());
    }
}
```

This resides in
the Eclipse-
Project
"Client"

A simple Example – 5

- Client should run with Version 2
and Provider (which should run on Version 1)



"Service v2" and
"Provider" are on
the build path of
"Client"

"Service v1" is still
on the build path of
"Provider"

- Output:
Calling Service: This is Service Version 2
Calling Provider: This is Service Version 2

"Service v1" is
not used!

Motivation – Modularization

- Recycling old ideas that were raised in the early seventies by, among others, David Parnas:
 - High Cohesion
 - Low Coupling

"On the criteria to be used in decomposing systems into modules"
Communications of the ACM, December 1972



David Parnas.
Source: Wikipedia

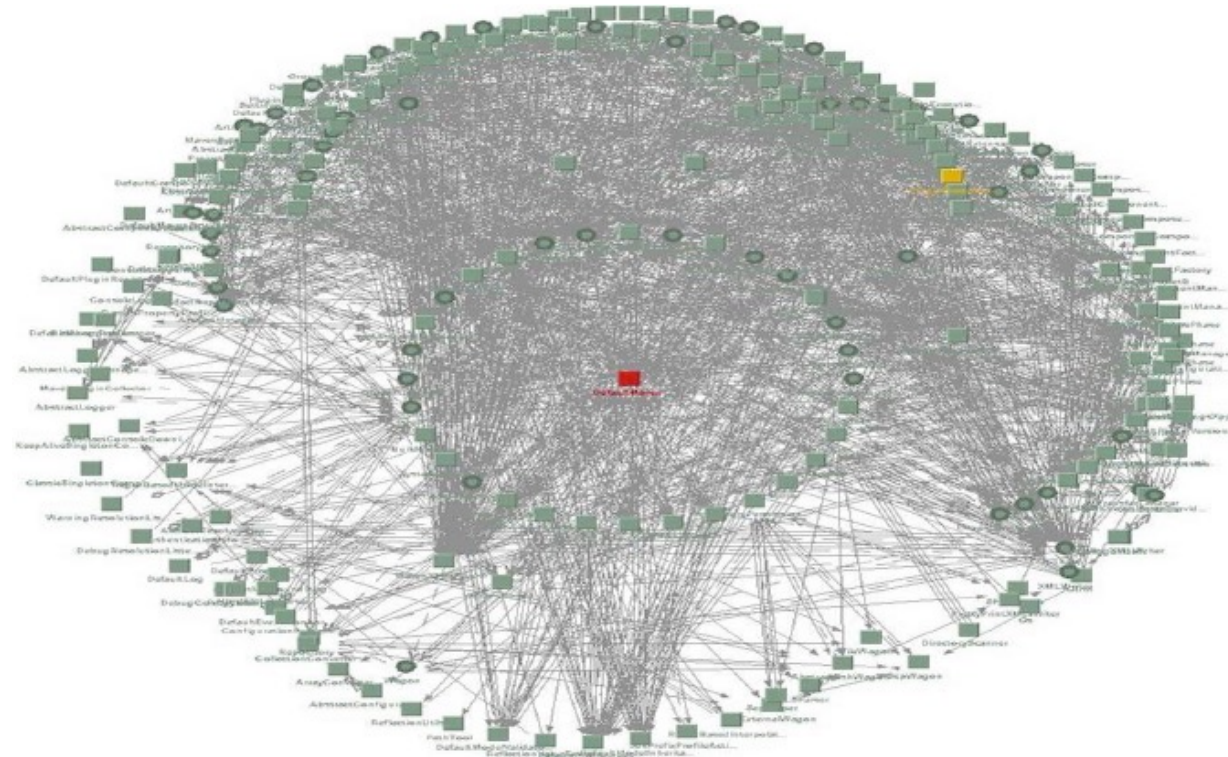
Motivation – Modularization (Cont.)

- Modularization minimizes complexity by creating proper boundaries between the parts that compose a system
- Properly modularized systems are easier to maintain and extend
 - Changes are more localized and affect less of the overall system.

**MANAGER: SO THERE WAS 3
BUGS BEFORE YOU STARTED AND
4 AFTER YOU FIXED ONE OF THEM?**



Motivation – What happened with modularization in OO?

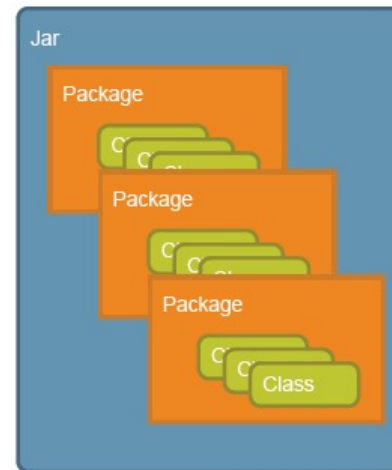


Motivation – What happened with modularization in OO?

- Focus in OO was on encapsulation of instance variables, which is some form of modularization, but granularity is too small
- OO systems become tangled webs quickly
 - Program knows its own structure
- Patterns like SOA, Factories, Dependency Injection, Inversion of Control are trying to minimize the consequences of OO and its lack of modularization.

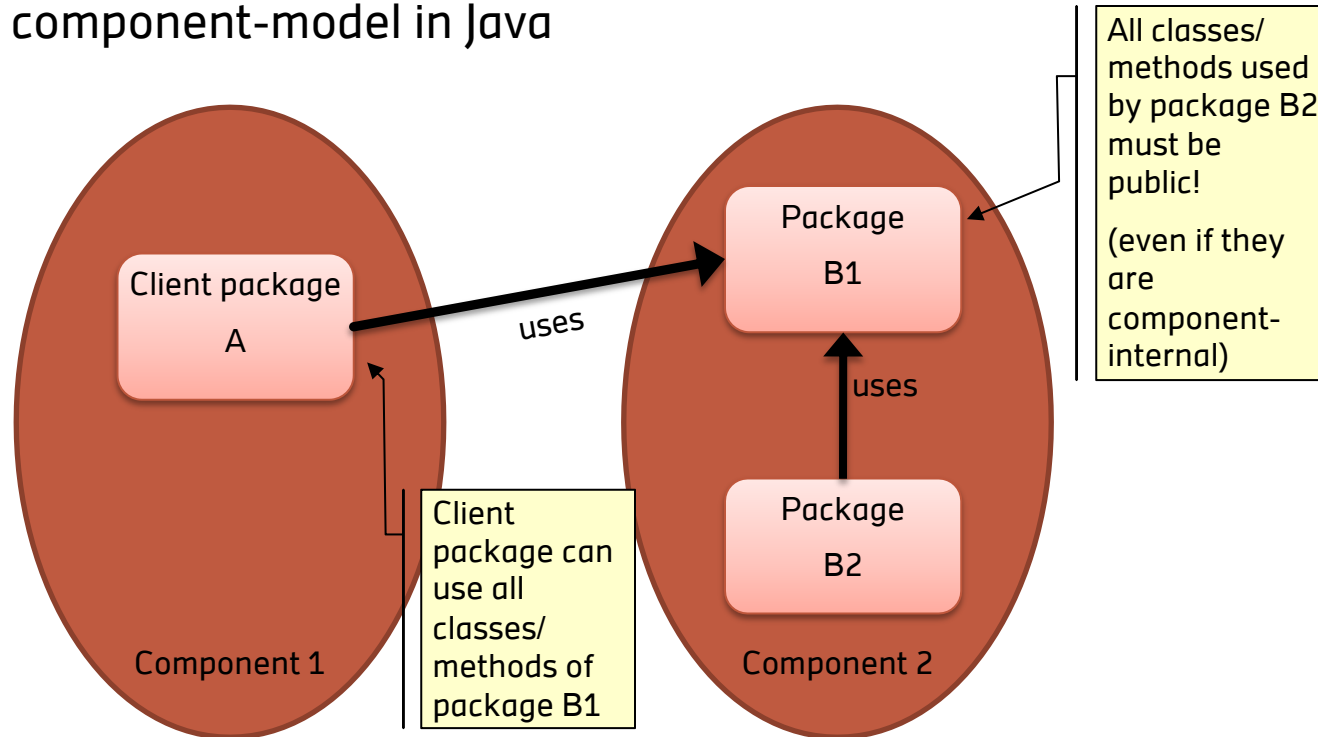
Motivation – Modularization in Java Apps – Visibility

- **Java Platform Modularity**
 - Classes encapsulate data
 - Packages contain classes
 - Jars contain packages
- **Visibility Access**
 - private, package private, protected, public
- Packages look hierarchical, but are not
- Jars have no modularization characteristics.



Motivation – Modularization in Java Apps – Visibility

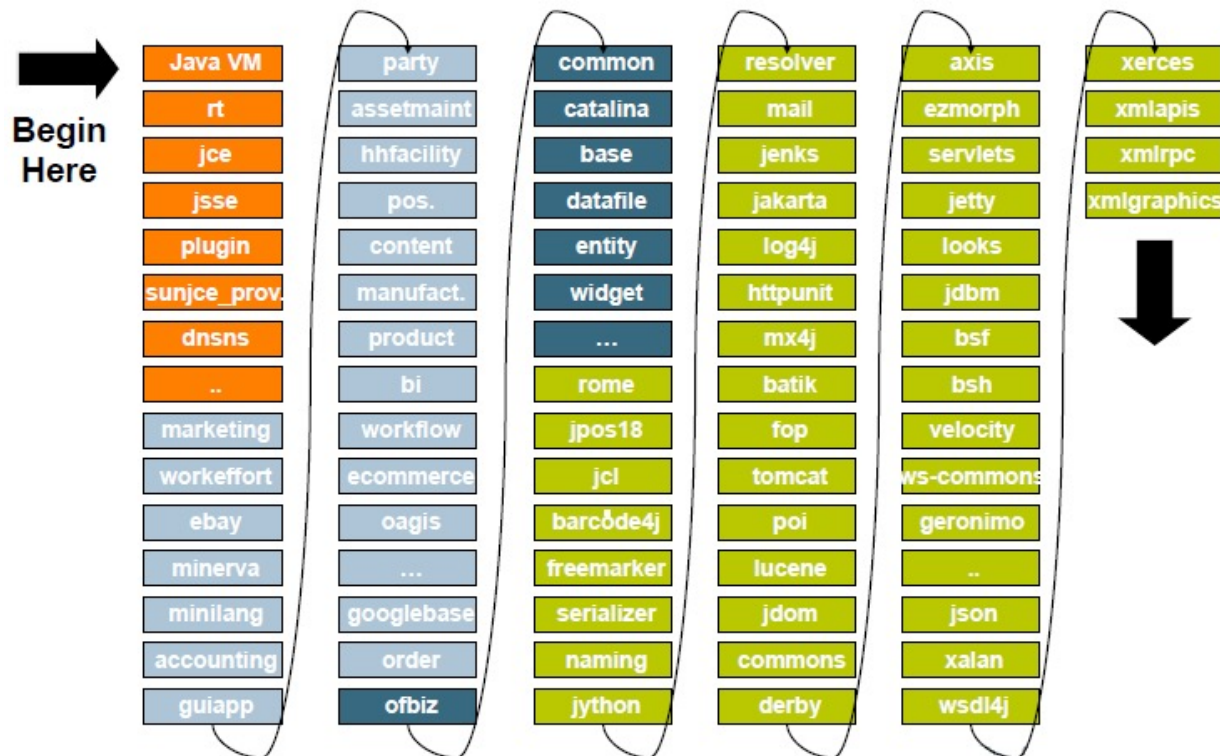
➤ Missing component-model in Java



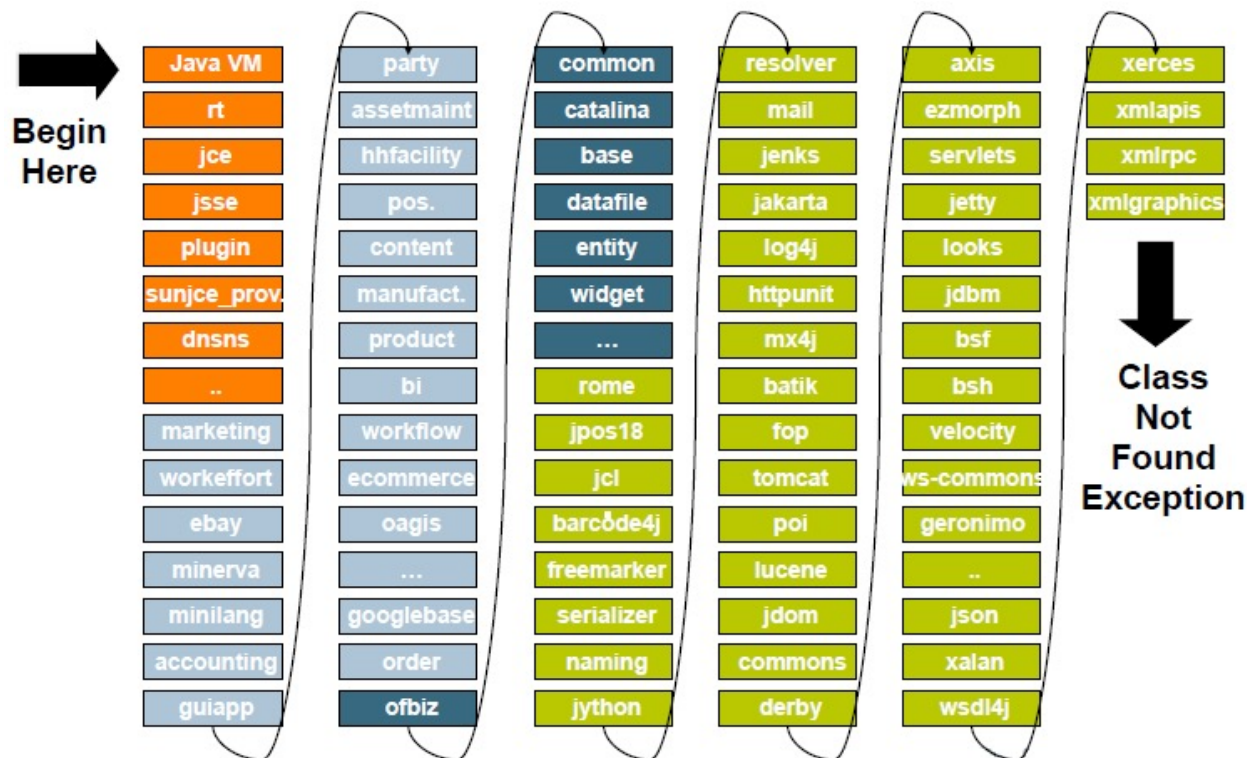
Motivation – Modularization in Java Apps – Runtime

- **Ever-growing and indivisible Java Runtime**
 - The Java runtime has always been growing in size. But before Java 8 there was no way to install a subset of the JRE; all Java installations were distributed with libraries for such APIs as XML, SQL and Swing, whether you needed them or not
 - While this may not be terribly significant for medium sized computing devices (for example desktop PCs or laptops) it is very significant for small devices like routers, TV-boxes, cars, etc. where Java is used
 - With the current trend of containerization, it also gains new relevance on servers, where reducing an image's footprint will reduce costs
- **Java 8 brought compact profiles (named compact 1 to compact3), which define three subsets of Java SE**
 - These alleviated the problem somewhat but only in restricted cases, and the profiles are too rigid to cover all current and future needs for partial JREs.

Motivation – Modularization in Java Apps – Classpath



Motivation – Modularization in Java Apps – Classpath



Motivation – Modularization in Java Apps – Classpath

➤ Unexpressed Dependencies

- A JAR cannot express which other JARs it depends on in a way that the JVM will understand. Users are hence left to identify and fulfill the dependencies manually, by reading the documentation, finding the correct projects, downloading the JARs and adding them to the project
- Then there are optional dependencies, where a JAR might only require another JAR if the user wants to use certain features. This complicates the process further
- The Java runtime will not detect an unfulfilled dependency until it is actually required. This will lead to a **NoClassDefFoundError** crashing the running application
- Build tools like Maven help to solve this problem.

Motivation – Modularization in Java Apps – Classpath

➤ Transitive Dependencies

- For an application to work it might only need a handful of libraries. Each of those in turn might need a handful of other libraries, and so on. As the problem is compounded it becomes exponentially more labor-intensive and error prone

- Again, this is helped by build-tools.

Motivation – Modularization in Java Apps – Classpath

➤ Shadowing

- Sometimes different JARs on the classpath contain classes with the same fully-qualified name, for example when they are two different versions of the same library
- Since classes will be loaded from the first JAR on the classpath that contains them, that variant will "shadow" all others and make them unavailable
- If the variants differ semantically, this can lead to anything from too-subtle-to-notice-misbehavior to havoc-wreaking-errors
 - Even worse, the form in which this problem manifests itself can seem non-deterministic
 - It depends on the order in which the JARs are listed on the classpath
 - This may well differ across different environments, for example between a developer's IDE and the production machine where the code will eventually run.

Motivation – Modularization in Java Apps – Classpath

➤ Version Collisions

- This problem arises when two required libraries depend on different versions of a third library
- If both versions are added to the classpath, the behavior will be unpredictable:
 - First, because of the shadowing problem, classes that exist in both versions will only be loaded from one of them
 - Worse, if a class that exists in one but not the other is accessed, that class will be loaded as well. Code calling into the library will hence find a mix of the two versions
- At best, the library code might fail loudly with a **NoClassDefFoundError** if it tries to access code that does not exist in a loaded class
- In the worst case, where versions only differ semantically, actual behavior may be subtly changed introducing hard-to-find bugs
- Identifying this as the source of unexpected behavior can be hard. Solving it directly is impossible.

Summary of the problems

- Granularity of classes and packages is too small for real world applications
 - Jars provide packaging but can not be used to restrict access
 - Every public class is visible to every other class
 - Severe problems like shadowing
 - Multiple Jars have classes in the same package
 - Often unintended
 - No versioning support
 - Order on CLASSPATH defines chosen version
 - Single version of a class in the VM
- Java up to version 8 is missing a component-model!

Overview

- Motivation
- Possible solutions
- OSGi
- Java Platform Module System (JPMS).

Solutions for the missing component model (1)

- Tools we covered in the lecture QA architecture
 - e.g. Sonargraph
 - Integration into the IDE, e.g. Eclipse
 - Immediate feedback to the programmer
- Checkstyle – ImportControl
 - Checks if the code complies with the given rules
 - Integration into the build-process
- Problem
 - Those tools only check dependencies during compile-time
 - Reflection-calls are not covered
 - Also not covered:
 - Classpath problem
 - Shadowing problem
 - Version problem.

Solutions for the missing component model (2)

➤ OSGi

- A dynamic module system for Java
 - we will give an overview to that here

➤ Java Platform Module System (JPMS)

- Built in since Java 9
 - main focus in this lecture.

Overview

- Motivation
- Possible solutions
- **OSGi**
- Java Platform Module System (JPMS).

What is OSGi?

➤ OSGi

- Open Services Gateway initiative (lower case 'i')
- A dynamic module system for Java
- Universal Middleware
- A SOA-in-JVM runtime
- Class-loaders with dependency-check

➤ Formally

- The OSGi Alliance was the non-profit organization behind it
- In 10/2020 OSGi was handed over to the Eclipse Foundation
- Since then, OSGi is developed by the Eclipse OSGi Working Group
- The output are the OSGi specifications

➤ The 3 main keywords

modules, dynamics and services.

What is Server-Side Eclipse (SSE)?

- Recognition... that many of the features that have made RCP (Rich Client Platform) successful are also applicable in server-side contexts
 - Standardized component model (OSGi)
 - Pervasive extensibility – Extension Registry
 - Runtime provisioning
- Integration... with existing server-side infrastructure and technologies
 - Java EE Application Servers
 - ALL major Java EE servers are based on OSGi: IBM WebSphere, JBoss, Spring, Oracle WebLogic, Sun Glassfish, SAP Netweaver
 - Servlets and JSPs, Application Frameworks...

History

➤ Available online

➤ <https://www.osgi.org/developer/specifications/>



1999	JSR-8 submitted & withdrawn OSGi alliance founded
2000	OSGi R1: Set-top boxes, DSL modems
2001	OSGi R2: Security improvements, remote management
2003	OSGi R3: Automotive and entertainment systems Eclipse switches from old component model to OSGi
2004	RCP movement starts Eclipse 3.0 ships, based on OSGi R3
2005	OSGi R4: Mobile devices, additional automotive support, Eclipse contributions. Eclipse 3.1 ships, based on OSGi R4
2006	Changes formally reviewed and committed Eclipse 3.2 ships Support for Servlet 2.4 and Jetty based HTTP service Server-side work graduates
2007	JDP 2.0 support graduates. OSGi R4, Version 4.1 released Eclipse 3.3 ships Founding of the Enterprise Experts Group
2008	Eclipse 3.4 ships with new provisioning infrastructure Widespread adoption of server vendors
2009	OSGi R4, Version 4.2 released
2010	OSGi R4, Version 4.2 Enterprise Specification released
2011	OSGi R4, Version 4.3 released
2012	OSGi R5, Version 5.0 released
2014	OSGi R6, Version 6.0 released
2018	OSGi R7, Version 7.0 released
2020	OSGi R8, Version 8.0 released

Bottom line: It's more
then 20 years old -
mature, solid

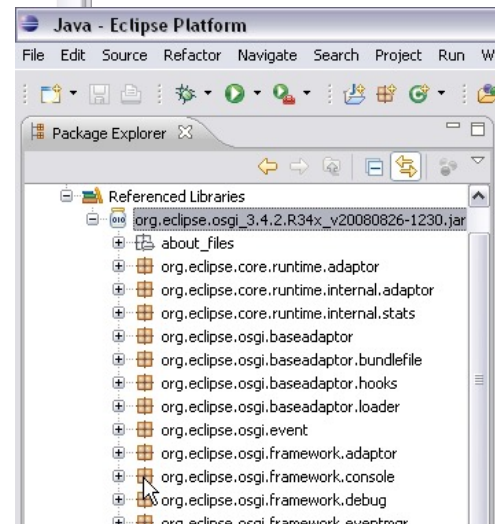
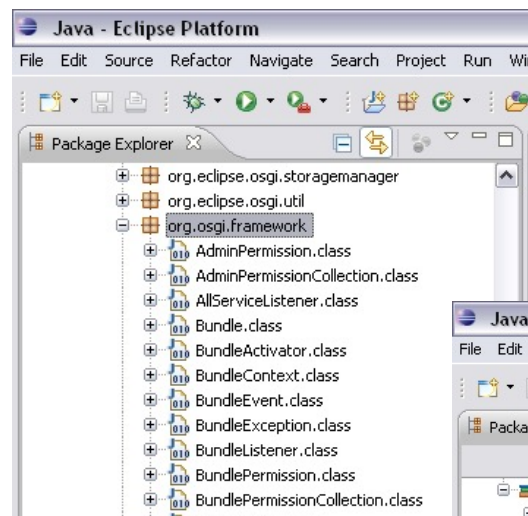
Eclipse Equinox

- Equinox is the reference-implementation of the OSGi core framework specification
- Additionally, it contains a set of bundles that implement various optional OSGi services and other infrastructure for running OSGi-based systems
- Other implementations
 - Apache Felix
 - Knopflerfish
 - ProSyst Software mBedded Server.



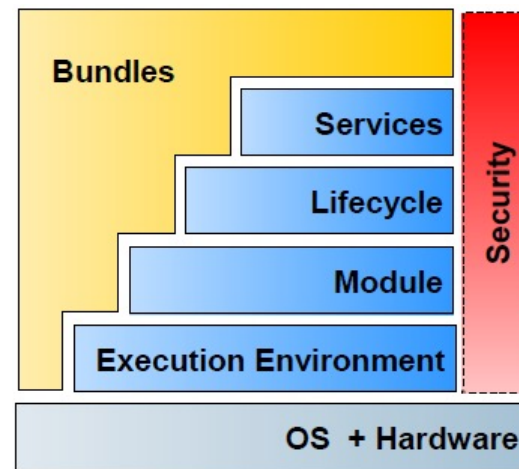
Eclipse Equinox – Packages

- Equinox JAR – OSGi packages
 - Implementation of the core framework specification
 - Includes the standard OSGi API and the Equinox implementation
 - One JAR file, <1MB, just a few dozen interfaces and classes!
- Equinox JAR – Eclipse packages
 - Implementation of optional OSGi services and other extensions
 - Note: the Equinox implementation contains some non-standard extensions.



OSGi layers

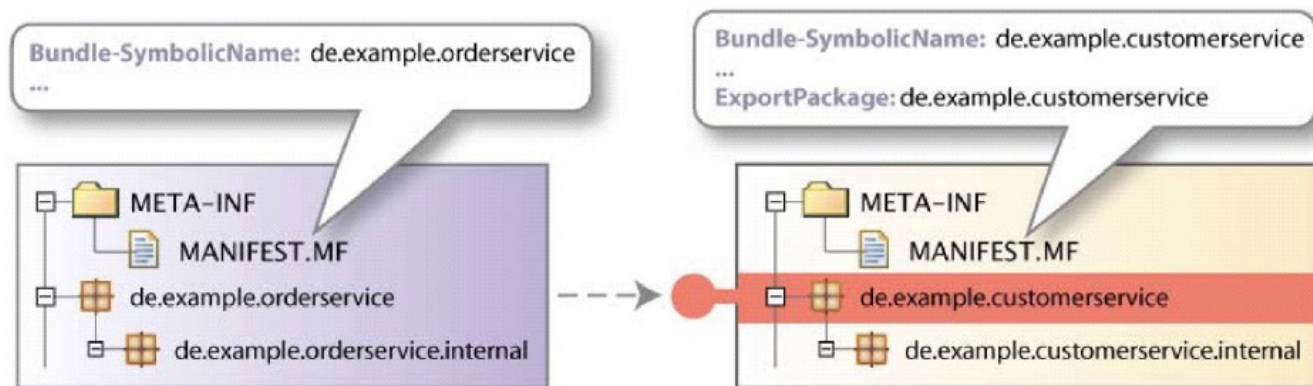
- **Execution Environment**
 - Describes the capabilities of Java VMs
- **Module layer**
 - Defines the component (plug-in) model with dependencies and versioning
- **Life Cycle layer**
 - Provides the lifecycle-functionality for bundles
- **Service layer**
 - Provides a central service repository and functions to use it.



Source: Introduction to Eclipse Equinox and OSGi,
Thomas Watson, Simon Kaegi, Webinar, 2007-11-27

Module layer

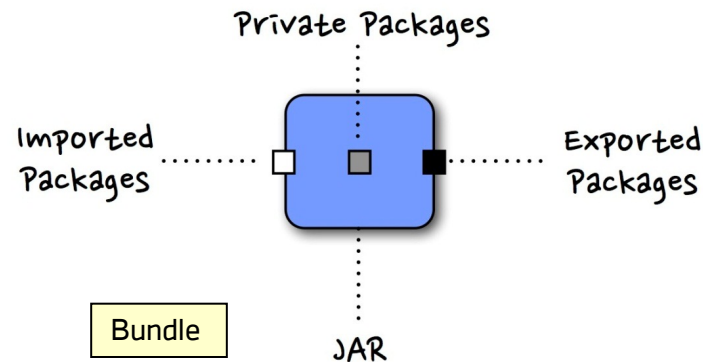
- OSGi defines a component model that was badly missing in Java
 - The components are called bundles (or plug-ins)
 - Bundles are basically conventional JAR files with additional metadata that is used by the runtime.



Source: Martin Lippert, EPL 1.0, 2007

Module layer - features

- Class loading dispatch based on package name
 - Prevents many problems with split packages
 - Faster class loading for large systems
- Allows multiple versions of the same class in one VM
 - Class spaces
- Jars can contain exported packages or Jar private packages
- Bundle == Jar
 - Manifest contains metadata.



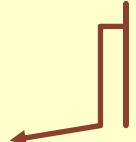
Example 1

➤ Our first bundle HelloActivator.java

```
import org.osgi.framework.*;
public class HelloActivator implements BundleActivator {
    public void start(BundleContext context) {
        System.out.println("Hello first bundle!");
    }
    public void stop(BundleContext context) {
        System.out.println("Goodbye first bundle!");
    }
}
```

➤ Manifest file MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloWorld
Bundle-Activator: HelloActivator
Bundle-SymbolicName: HelloWorld
Bundle-Version: 1.0.0.qualifier
Import-Package: org.osgi.framework
```



This empty blank
line at the end is
important!

Example 1 (Cont.)

➤ Compile file and create Jar-file

```
> Javac -classpath  
  /Java/eclipse/plugins/org.eclipse.osgi_<version>.Jar  
  HelloActivator.Java  
  
> Jar -cfm HelloWorld.Jar MANIFEST.MF HelloActivator.class
```

➤ Run OSGi runtime and install bundle

```
> Java -Jar /Java/eclipse/plugins/org.eclipse.osgi_<version>.Jar -console  
  
osgi> ss  
  
Framework is launched.  
  
id      State      Bundle  
0       ACTIVE     org.eclipse.osgi_<version>  
  
osgi> install file:HelloWorld.Jar  
Bundle id is 2
```

Example 1 (Cont.)

➤ Start, stop and uninstall bundle

```
osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE      org.eclipse.osgi_<version>
2       INSTALLED   HelloWorld_1.0.0

osgi> start 2
Hello first bundle!

osgi> stop 2
Goodbye first bundle!

osgi> uninstall 2
osgi> ss

Framework is launched.

id      State      Bundle
0       ACTIVE      org.eclipse.osgi_<version>
```

Module layer – dependency-declaration (1/2)

➤ Dependencies between bundles

- Dependency-declarations are part of the manifest-file of each bundle
- Bundles that export functionality must declare the packages including the version they're providing

```
Export-Package: de.hftstuttgart.swe.something;version="1.0.0"
```

Module layer – dependency-declaration (2/2)

➤ Dependencies between bundles

- Bundles that want to use functionality from other bundles have to declare the packages they want to use

```
Import-Package: de.hftstuttgart.swe.something
```

➤ Declaration of an allowed version-range

```
Import-Package: de.hftstuttgart.swe.something;version="[1.0.0,2.0.0)"
```

➤ Usage of multiple packages is also possible

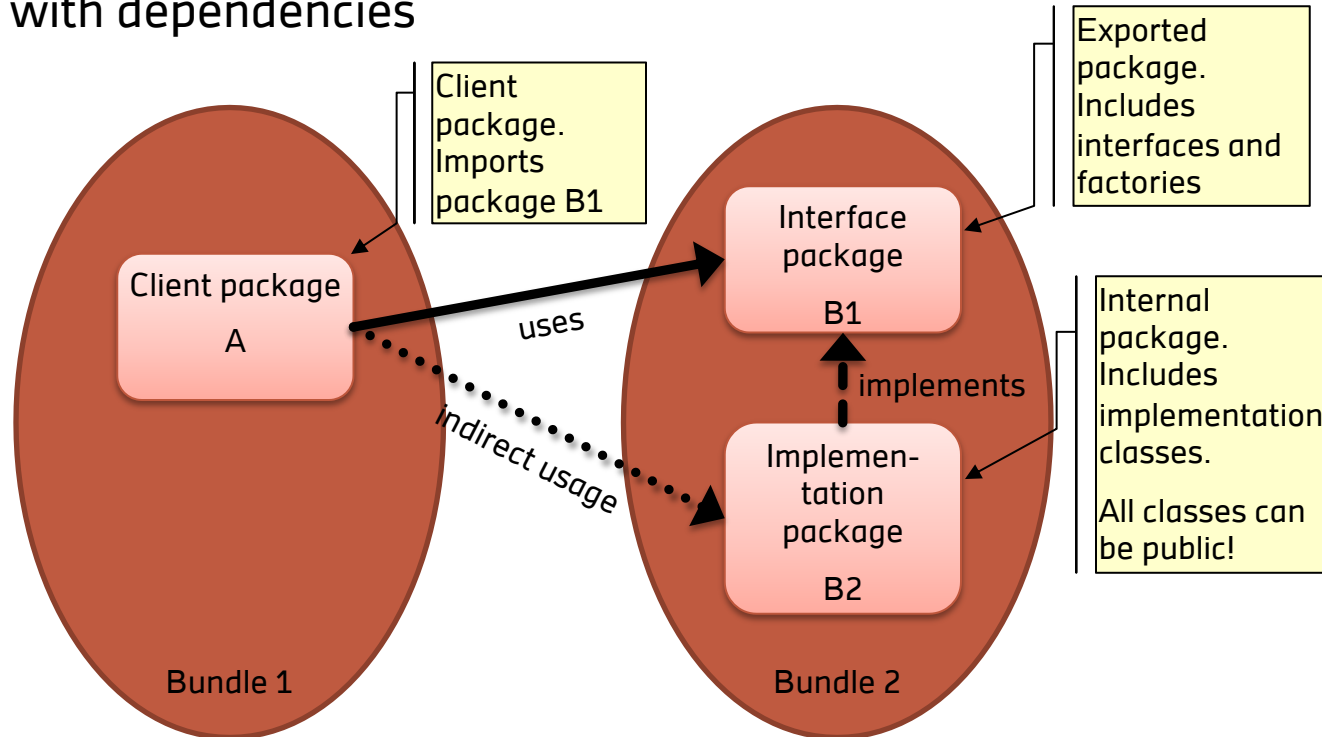
```
Import-Package: org.osgi.framework,  
de.hftstuttgart.swe.something="[1.0.0,2.0.0)",  
de.hftstuttgart.swe.somethingelse="[1.0.0,2.0.0)"
```

➤ Examples of version ranges

Example	Predicate
[1.2.3, 4.5.6)	1.2.3 <= x < 4.5.6
(1.2.3, 4.5.6]	1.2.3 < x <= 4.5.6
1.2.3	1.2.3 <= x
nothing = [0.0.0, ∞)	

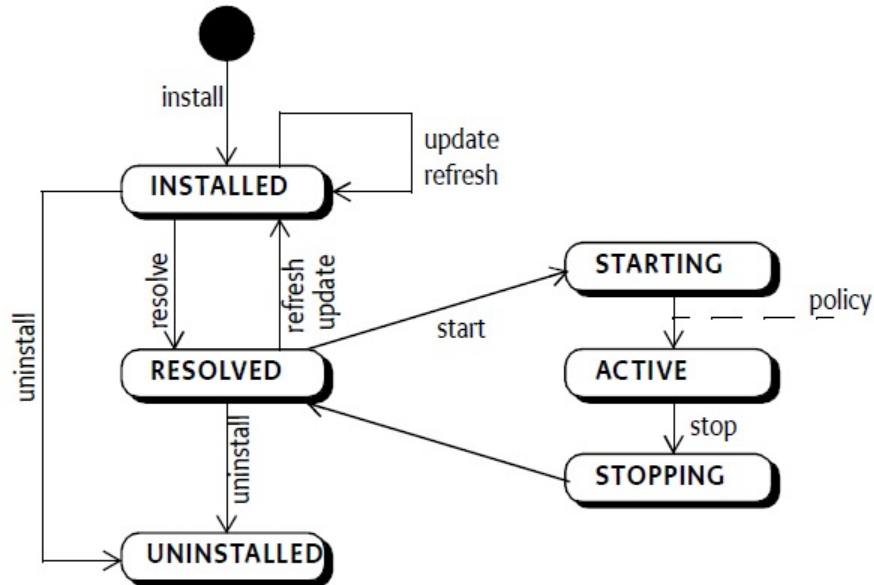
Module layer – best practice

➤ Bundle with dependencies



Lifecycle layer

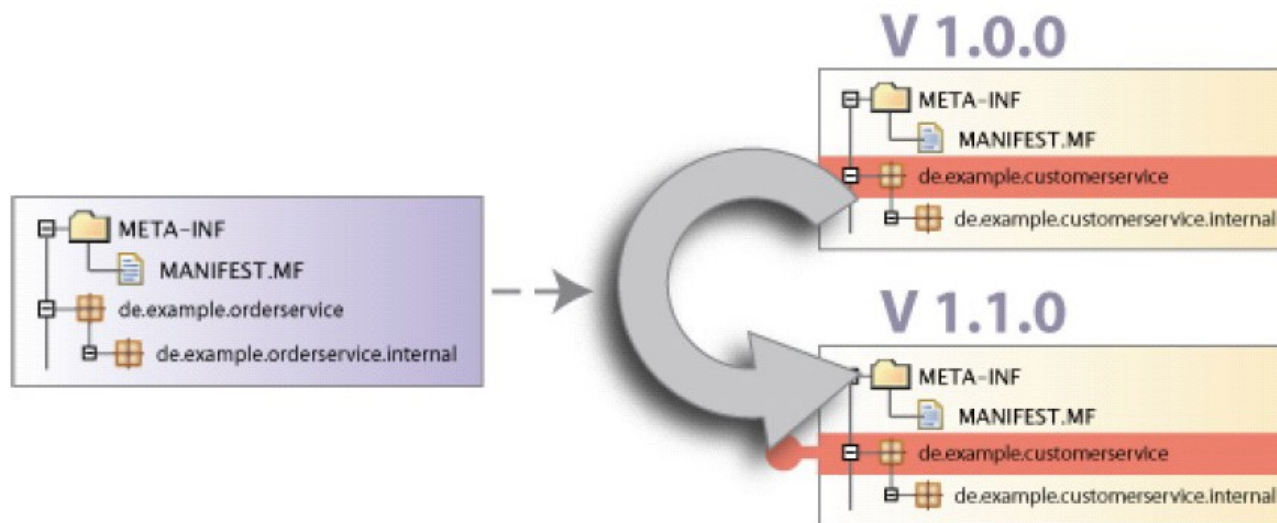
- Install, Uninstall
- Start, Stop
- Update



Source: OSGi R4.2 Core
Specification, chapter. 4.4.2

Lifecycle layer – dynamic runtime

- OSGi specifies a runtime environment in which the key elements, bundles and services, can be added and removed at runtime.

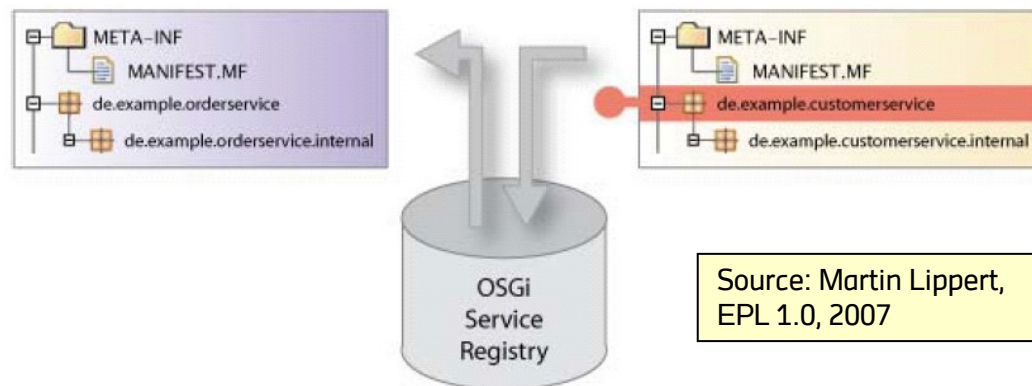


Source: Martin Lippert, EPL 1.0, 2007

Service layer - Goals

- Services provide a very loose coupling between modules
 - Services are used as input/output ports, they *should be the only links between bundles*
 - Key is to find connections between modules and map them to services
- Best practices:
 - Try to use standard services (OSGi)
 - Use Declarative Services (DS): declarative registration and usage of services

➤ Service registry



Source: Martin Lippert,
EPL 1.0, 2007

Summary, benefits of OSGi

- "Classpath hell" is over
- Module layer gives you more then 50% benefit of OSGi
 - You can, but do not have to use other layers
 - Enables concurrent development in large teams (IBM WAS)
 - Parallel versions are easy, especially important in SaaS and SOA
- Lifecycle layer enables zero-downtime operations.

**Until Java 9 OSGi was the best
modularization-framework for Java**

Overview

- Motivation
- Possible solutions
- OSGi
- Java Platform Module System (JPMS).

JPMS - Overview

- Standard-module-system for Java 9
 - Working title was "Jigsaw"
- Based on 6 JEPs and one JSR

JEP = Java Enhancement Proposal
JSR = Java Specification Request

JEP 200: Modular JDK
JEP 201: Modular Source Code
JEP 220: Modular Run-Time Images
JEP 260: Encapsulate Most Internal APIs

JEP 261: Module System
JEP 282: jlink: The Java Linker
JSR 376: Java Platform Module System

- Accessibility 1995-2017

- `public`
- `protected`
- `<package>`
- `private`

- Accessibility 2017-
 - 'public' no longer means "accessible".

- `public to everyone`
- `public but only to specific modules`
- `public only within a module`
- `protected`
- `<package>`
- `private`

JPMS - Primary goals

- Make it easier for developers to construct and maintain libraries and large applications
- Improve the security and maintainability of Java SE Platform Implementations in general, and the JDK in particular
- Enable improved application performance
- Enable the Java SE Platform, and the JDK, to scale down for use in small computing devices and dense cloud deployments.

JPMS – Module

- A module in JPMS is a container with packages
- It includes resource-files, source code, and a module-descriptor with the name **module-info.java**
 - Typical structure of a Java module

```
src/  
  de.hftstuttgart.myModule /  
    module-info.java  
      de/hftstuttgart/myModule /  
        // my Java-files
```


JPMS – Module descriptor

➤ The module-descriptor has the name **module-info.java**

- This file must be located on the top-level package-level of the module
 - It is only allowed to put module-information in this file. No Java classes!
 - Only one module-declaration per module-descriptor is allowed

```
module de.hftstuttgart.myModule {  
}
```

- In the example above there are no statements
 - This means that no other module can access this module and this module also cannot access other modules
 - The principle of strong encapsulation is built-in by default!
- No module-descriptor is necessary if...
 - a Jar-file is part of the module-path in the command-line option
 - The module-descriptor is generated for a Jar-file using the tool **jdeps**.

Declarations in the module-descriptor

Declaration	Description
requires <name of the module>	Describes, on which module the actual module depends
exports <name of the package> (to <name of the module>)	Describes, which packages from the actual module are exported outside the module
uses <type of service>	Defines, that the actual module consumes an instance of the service
provides <name of the service> with <name of the implementation of the service>	Defines that the actual module provides a service using the implementation of the service
opens <name of the package>	Provides the package with the given name for deep reflection during runtime → Give an external library like e.g. Spring access to also the private classes (visibility and access) of the package

JPMS – Declare dependencies

- The **requires**-declaration is used inside the module-declaration to define the modules, that are required by the actual module to fulfil its dependencies
 - Syntax: **requires** <name of module>;
- For each module, a separate requires-declaration must be defined

```
// module descriptor of module "de.hftstuttgart.module1"  
module de.hftstuttgart.module1 {  
    requires de.hftstuttgart.module2;  
    requires de.hftstuttgart.module3;  
}
```

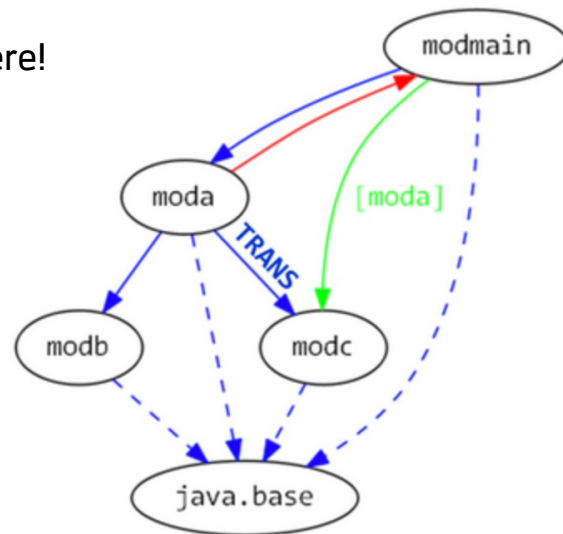
- Module **de.hftstuttgart.module1** has dependencies to the modules **de.hftstuttgart.module2** and **de.hftstuttgart.module3**
- That means that the module **de.hftstuttgart.module1** uses types that are part of the module **de.hftstuttgart.module2** or **de.hftstuttgart.module3**

JPMS – Transitive dependencies

- Dependencies are not transitive by default
- If a method in class **A** in module **moda** returns an object of class **C** in module **modc**, the caller of that method in the module **modmain** also needs access to **modc**
 - But dependencies of **moda** should be encapsulated there!
 - It would be very ugly, if every "user" of **moda** must declare also all dependencies of **moda** again

```
// module descriptor of module "moda"  
module moda{  
    requires modb;  
    requires transitive modc;  
}
```

- A transitive-dependency automatically gives access to dependent modules
- A chain of transitive-dependencies is possible.



JPMS – Export packages

- The **exports**-declaration has the task to export a package for compiletime and runtime
 - A package that is not exported is not available for other modules
- A module exports nothing per default. No parts of a module are accessible from other modules per default
 - Syntax: **exports** <name of package>;

```
// module-descriptor of module "de.hftstuttgart.module2"  
module de.hftstuttgart.module2 {  
    exports de.hftstuttgart.module2.package1;  
}
```

- The package `de.hftstuttgart.module2.package1` is exported
- Given that the module `de.hftstuttgart.module1` requires the module `de.hftstuttgart.module2`, the package `de.hftstuttgart.module2.package1` is accessible by the module `de.hftstuttgart.module1`.

JPMS – Accessibility

- Accessibility has changed completely with Java 9
 - Before Java 9 it was possible to have a class defined as public that was accessible from all other parts of the application
- In Java 9 three conditions must be fulfilled to allow accessibility
 - The first module must require the second module using the requires-declaration in the module-descriptor
 - The second module must export the package that the first module requires
 - Types in the exported package must be declared as public
- Accessibility-rules are checked in Java 9 both during compilation and during runtime
 - An error of the type `IllegalAccessError` is thrown during runtime, if the accessibility-rules are violated
 - Accessibility-rules are enforced directly in/by the JVM.

JPMS – Accessibility

➤ Accessibility- rules in Java 9

module declared as required	package exported	type of modifier	accessible in other module
yes	yes	Public	yes
yes	yes	Protected	no
yes	yes	(default)	no
yes	yes	Private	no
yes	no	Public	no
yes	no	Protected	no
yes	no	(default)	no
yes	no	Private	no
no	yes	Public	no
no	yes	Protected	no
no	yes	(default)	no
no	yes	Private	no
no	no	Public	no
no	no	Protected	no
no	no	(default)	no
no	no	Private	no

JPMS – module-path

- JPMS introduces a new concept to replace the classpath: the module-path, a collection of directories that contain modules
 - In contrast to the module-path the classpath is a collection of jar-files
 - The module-path allows the specification of modules instead of jar-files
 - If a module is available at different places, the first occurrence is used
- The module-path is used by the compiler to find modules
 - It can also be mixed with the classpath, that is still present in Java 9
- Different module-paths in JPMS
 - Compilation-module-path (`module-source-path`) specifies the definitions of the modules and is related only to the modules that are not already compiled
 - Application-module-path (`module-path`) specifies the directories that contain the already compiled modules.

JPMS – Compile files

- Compilation in Java 9 is only different if using modules
 - An additional process for module-resolution is started during the compilation
 - This process calculates the minimal amount of modules that are necessary to fulfil all dependencies in the application

```
javac --module-source-path --module-path -d
```

- Syntax for compilation using the module-path
 - Option `-d` specifies the output-directory that will contain the compiled files
 - Option `--module-source-path` specifies the definitions of the modules and is related to the modules that are not already compiled
 - Option `--module-path` specifies the directories that contain the already compiled modules.

JPMS – Execute application

- Execution of an application in Java 9 only is different, if the module-path is used
 - The main-module is loaded, all dependencies are resolved and the mainclass is executed

```
java --module-path <list of directories with modules> --module  
<name of module>/<name of mainclass>
```

- Syntax for running an application
 - Option `--module-path` (or short `-p`) is the module-path
 - Option `--module` (or short `-m`) gets as parameter the module-name together with the mainclass.

Comparison OSGi - JPMS

- JPMS solves the most of what was missing in Java – a real module system
 - It is simpler than OSGi
 - It is integral part of Java
 - The JDK itself is based on JPMS
- OSGi has much more features , e.g. dynamic unloading and loading of modules, service-registry, etc.
 - But these features often are not necessary, especially in microservice-architectures.

Literature

➤ OSGi

- OSGi Core Specification, Release 7,
- 2018, <https://www.osgi.org/release-7-1/>

➤ BJ Hargrave, Peter Kriens

- Converting (Large) Applications to OSGi™
- JavaOne 2008, TS-5122

➤ Frank Gerhardt

- Eclipse Equinox - The Dynamic Server Platform based on OSGi
- ObjectForum, 12/2008, Karlsruhe

➤ Neil Barlett

- Getting Started with OSGi - Tutorial
- <http://neilbartlett.name/blog/osgi-articles/>

➤ Nicolai Parlog

- Project Jigsaw is Really Coming in Java 9
- 2015, <http://www.infoq.com/articles/Project-jigsaw-Coming-in-Java-9>