JDK 9 und die modulare Plattform Jigsaw

CREATE
THE
FUTURE

Wolfgang Weigend
Sen. Leitender Systemberater
Java Technology and Architecture

JavaOne
ORACLE

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda

1. JDK 9 Status
2. Modularity
3. Jigsaw und die Werkzeuge
4. Participation
5. Ausblick und Zusammenfassung

JDK 9 Status

http://openjdk.java.net/projects/jdk9/

- The goal of this Project is to produce an open-source reference implementation of the Java SE 9 Platform, to be defined by a forthcoming JSR in the Java Community Process
- The schedule and features of this release are proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal
- **Schedule**

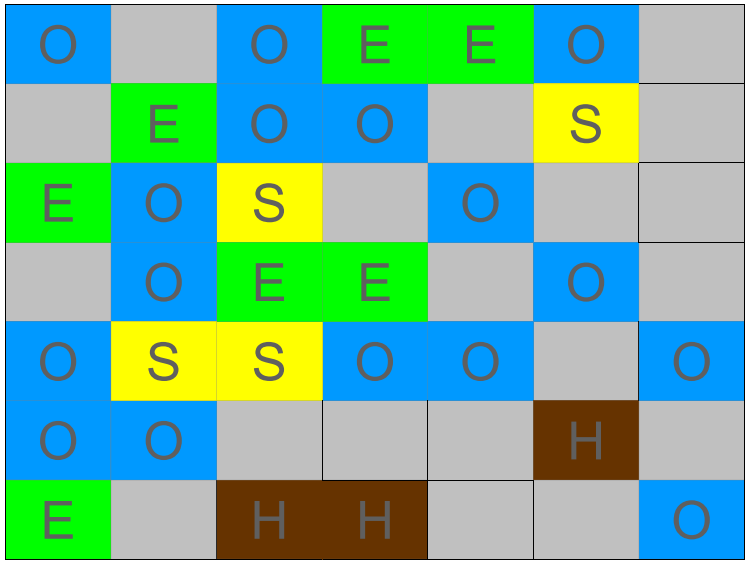
2016/05/26	Feature Complete
2016/12/22	Feature Extension Complete
2017/01/05	Rampdown Start
2017/02/09	All Tests Run
2017/02/16	Zero Bug Bounce
2017/03/16	Rampdown Phase 2
2017/07/06	Final Release Candidate
2017/07/27	**General Availability**

JDK 9 Status – 89 JEP's targeted to JDK 9

http://openjdk.java.net/projects/jdk9/

102: Process API Updates
110: HTTP 2 Client
143: Improve Contended Locking
158: Unified JVM Logging
165: Compiler Control
193: Variable Handles
197: Segmented Code Cache
199: Smart Java Compilation, Phase Two
200: The Modular JDK
201: Modular Source Code
211: Elide Deprecation Warnings on Import Statements
212: Resolve Lint and Doclint Warnings
213: Milling Project Coin
214: Remove GC Combinations Deprecated in JDK 8
215: Tiered Attribution for javac
216: Process Import Statements Correctly
217: Annotations Pipeline 2.0
219: Datagram Transport Layer Security (DTLS)
220: Modular Run-Time Images
221: Simplified Doclet API
222: jshell: The Java Shell (Read-Eval-Print Loop)
223: New Version-String Scheme
224: HTML5 Javadoc
225: Javadoc Search
226: UTF-8 Property Files
227: Unicode 7.0
228: Add More Diagnostic Commands
229: Create PKCS12 Keystores by Default
231: Remove Launch-Time JRE Version Selection
232: Improve Secure Application Performance
233: Generate Run-Time Compiler Tests Automatically
235: Test Class-File Attributes Generated by javac
236: Parser API for Nashorn
237: Linux/AArch64 Port
238: Multi-Release JAR Files
240: Remove the JVM TI hprof Agent
241: Remove the jhat Tool
243: Java-Level JVM Compiler Interface
244: TLS Application-Layer Protocol Negotiation Extension
245: Validate JVM Command-Line Flag Arguments
246: Leverage CPU Instructions for GHASH and RSA
247: Compile for Older Platform Versions
248: Make G1 the Default Garbage Collector
249: OCSP Stapling for TLS
250: Store Interned Strings in CDS Archives
251: Multi-Resolution Images
252: Use CLDR Locale Data by Default
253: Prepare JavaFX UI Controls & CSS APIs for Modularization
254: Compact Strings
255: Merge Selected Xerces 2.11.0 Updates into JAXP
256: BeanInfo Annotations
257: Update JavaFX/Media to Newer Version of GStreamer
258: HarfBuzz Font-Layout Engine
259: Stack-Walking API
260: Encapsulate Most Internal APIs
261: Module System
262: TIFF Image I/O
263: HiDPI Graphics on Windows and Linux
264: Platform Logging API and Service
265: Marlin Graphics Renderer
266: More Concurrency Updates
267: Unicode 8.0
268: XML Catalogs
269: Convenience Factory Methods for Collections
270: Reserved Stack Areas for Critical Sections
271: Unified GC Logging
272: Platform-Specific Desktop Features
273: DRBG-Based SecureRandom Implementations
274: Enhanced Method Handles
275: Modular Java Application Packaging
276: Dynamic Linking of Language-Defined Object Models
277: Enhanced Deprecation
278: Additional Tests for Humongous Objects in G1
279: Improve Test-Failure Troubleshooting
280: Indify String Concatenation
281: HotSpot C++ Unit-Test Framework
282: jlink: The Java Linker
283: Enable GTK 3 on Linux
284: New HotSpot Build System
285: Spin-Wait Hints
287: SHA-3 Hash Algorithms
288: Disable SHA-1 Certificates
289: Deprecate the Applet API
290: Filter Incoming Serialization Data
292: Implement Selected ECMAScript 6 Features in Nashorn
294: Linux/s390x Port
295: Ahead-of-Time Compilation
297: Unified arm32/arm64 Port
298: Remove Demos and Samples

JDK 9 - G1 Garbage Collector as the default



E Eden regions
S Survivor regions
O Old generation regions
H Humongous regions
Available / Unused regions

- GC pause-times have the largest impact on application performance, predictability and responsiveness
- One large contiguous heap space divided into many fixed size regions
  - Size can be 1 MB – 32 MB
  - Only GC that can scale up to multi-TB heap
- Each region can be assigned a unique eviction/compaction policy (Eden region, Survivor region, Humongous or Old region)
- Per region scalable collection process
- Allow optimized memory mapping between the OS and the JVM

JEP 222: jshell – Read-Eval-Print Loop

```
C:\projects\jdk-9>jshell
|  Welcome to JShell -- Version 9-ea
|  Type /help for help

-> /help
|  Type a Java language expression, statement, or declaration.
|  Or type one of the following commands:
|
|     /list [all|start|<name or id>]                              -- list the source you have typed
|     /edit <name or id>                                          -- edit a source entry referenced by name or id
|     /drop <name or id>                                          -- delete a source entry referenced by name or id
|     /save [all|history|start] <file>                            -- Save snippet source to a file.
|     /open <file>                                                -- open a file as source input
|     /vars                                                       -- list the declared variables and their values
|     /methods                                                    -- list the declared methods and their signatures
|     /classes                                                    -- list the declared classes
|     /imports                                                    -- list the imported items
|     /exit                                                       -- exit jshell
|     /reset                                                      -- reset jshell
|     /reload [restore] [quiet]                                   -- reset and replay relevant history -- current or previous (restore)
|     /classpath <path>                                           -- add a path to the classpath
|     /history                                                    -- history of what you have typed
|     /help [<command>|<subject>]                                 -- get information about jshell
|     /set editor|start|feedback|newmode|prompt|format|field ...  -- set jshell configuration information
|     /?                                                          -- get information about jshell
|     /!                                                          -- re-run last snippet
|     /<id>                                                       -- re-run snippet by id
|     /-<n>                                                       -- re-run n-th previous snippet
|

->
```

JEP 223: New Version-String Scheme (1)

Revise the JDK's version-string scheme: Project Verona

- It's long past time for a simpler, more intuitive versioning scheme.
- A *version number* is a non-empty sequence of non-negative integer numerals, without leading zeroes, separated by period characters
  - [1-9][0-9]*(\.(0|[1-9][0-9]*))*
- $MAJOR.$MINOR.$SECURITY
- A *version string* consists of a version number $VNUM, as described above, optionally followed by pre-release and build information
- This proposal drops the initial 1 element from JDK version numbers.
  - First release of JDK 9 will have the version number 9.0.0 rather than 1.9.0.0.

JEP 223: New Version-String Scheme (2)

New version-string format : Hypothetical Examples

Release type	Old format Long form	Old format Short form		New format Long form	New format Short form
Early Access	1.9.0-ea-b19	9-ea	->	9-ea+19	9-ea
Major	1.9.0-b100	9	->	9+100	9
CPU	1.9.0_5-b20	9u5	->	9.0.1+20	9.0.1
Minor	1.9.0_20-b62	9u20	->	9.1.2+62	9.1.2

JEP 223: New Version-String Scheme (3)

A simple JDK-specific Java API to parse, validate, and compare version strings

```
package jdk;

import java.util.Optional;

public class Version
    implements Comparable<Version>
{

    public static Version parse(String);
    public static Version current();

    public int major();
    public int minor();
    public int security();

    public List<Integer> version();
    public Optional<String> pre();
    public Optional<Integer> build();
    public Optional<String> optional();

    public int compareTo(Version o);
    public int compareToIgnoreOpt(Version o);

    public boolean equals(Object o);
    public boolean equalsIgnoreOpt(Object o);

    public String toString();
    public int hashCode();
}
```

Release Type	Proposed
Major (GA)	jdk-9+100
Minor #1 (GA)	jdk-9.1.2+27
Security #1 (GA)	jdk-9.0.1+3

Migrating to Oracle JDK 9 - Migration Guide (1)

https://docs.oracle.com/javase/9/migrate/

- **How to proceed as you migrate your existing Java application to JDK 9**
  - Every new Java SE release introduces some binary, source and behavioral incompatibilities with previous releases
  - The modularization of the Java SE Platform brings many benefits but also many changes
  - Code that uses only official Java SE Platform APIs and supported JDK-specific APIs should continue to work without change
  - Code that uses certain features or JDK-internal APIs may not run or may give different results
- **Prepare for Migration**
  - Get the JDK 9 Early Access Build
  - Run Your Program Before Recompiling
  - Update Third-Party Libraries
  - Compile Your Application
  - Run jdeps on Your Code

Migrating to Oracle JDK 9 - Migration Guide (2)

https://docs.oracle.com/javase/9/migrate/

- **Beware of changes that you may encounter as you run your application**
  - Changes to the Installed JDK/JRE Image
  - Removed APIs
  - Deployment
  - Changes to Garbage Collection
  - Removed Tools
  - Removed macOS-specific Features

Migrating to Oracle JDK 9 - Migration Guide (3)

Removed Tools

- **JavaDB**, which was a rebranding of Apache Derby, is not included in JDK 9
- **JVM Tools Interface hprof agent** library (libhprof.so) has been removed
  - The hprof agent was written as demonstration code for the **JVM Tool Interface** and not intended to be a production tool. The useful features of the hprof agent have been superseded by better tools in the JDK
- The **jhat tool** was an experimental, unsupported heap visualization tool added in JDK 6. Superior heap visualizers and analyzers have been available for many years
- The **launchers java-rmi.exe** from Windows and **java-rmi.cgi** from Linux and Solaris have been removed
- The **IIOP transport** support from the **JMX RMI Connector** along with its supporting classes have been removed in JDK 9
- **Windows 32 Client VM is dropped** and only a server VM is offered in JDK 9
- **Visual VM** removed
  - Visual VM is a tool that provides information about code running on a Java Virtual Machine. It was provided with JDK 6, JDK 7, and JDK 8
  - Visual VM is not bundled with JDK 9. If you would like to use Visual VM with JDK 9, you can get it from the Visual VM open source project site

Modularity Landscape

A little bit of background

- **Java Platform Module System**
  - JSR 376 which is targeted for Java SE 9
- **Java SE 9 Platform Umbrella JSR**
  - JSR 379
  - Will own the modularization of the Java SE APIs
- **OpenJDK Project Jigsaw**
  - Reference Implementation for JSR 376
  - JEP 200, 201, 220, 260, 261, 282

JSR 376: Java Platform Module System

An approachable yet scalable module system for the Java Platform

- Provide a means for developers and libraries to define their own modules
- Reflection API's for module information
- Integration with developer tools (Maven, Gradle, IDE's)
- Integration with existing package managers (e.g., RPM)
- Dynamic configuration of module graph (e.g., for Java EE containers)
- Current documents, code, & builds
  - ❑Requirements
  - ❑The State of the Module System (design overview)
  - ❑Initial draft JLS and JVMS changes
  - ❑Draft API specification (diffs relative to JDK 9)
    - java.lang.Class
    - java.lang.ClassLoader
    - java.lang.reflect.Module
    - java.lang.module
  - ❑Issue summary
  - ❑RI prototype: Source, binary

Projekt Jigsaw

JDK Enhancement Proposal's (JEP's)

- JSR 376 Java Platform Module System
- JEP 200: The 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
- OpenJDK Jigsaw Early Access builds are available
  - JDK 9 Early Access with Project Jigsaw, build 167
  - http://jdk.java.net/jigsaw/

JEP 200: The Modular JDK (1)

Goal: Define a modular structure for the JDK

- Make minimal assumptions about the module system that will be used to implement that structure.
- **Divide the JDK into a set of modules** that can be combined at **compile time**, **build time**, **install time**, **or run time** into a variety of configurations including, **but not limited to**:
  - Configurations corresponding to the full Java SE Platform, the full JRE, and the full JDK;
  - Configurations roughly equivalent in content to each of the Compact Profiles defined in Java SE 8; and
  - Custom configurations which contain only a specified set of modules and the modules transitively required by those modules.

JEP 200: The Modular JDK (2)

Module System Assumptions: A module …

- can contain class files, resources, and related native and configuration files.
- **has a name**.
- can depend, by module name, upon one or more other modules.
- **can export all of the public types in one or more of the API packages that it contains, making them available to code in other modules depending on it**
- can restrict, by module name, the set of modules to which the public types in one or more of its API packages are exported. (sharing internal interface)
- **can re-export all of the public types that are exported by one or more of the modules upon which it depends**. (support refactoring & aggregation)
  - ***A module is a set of packages with classes & interfaces***
  - ***The module metadata is in module-info.class***

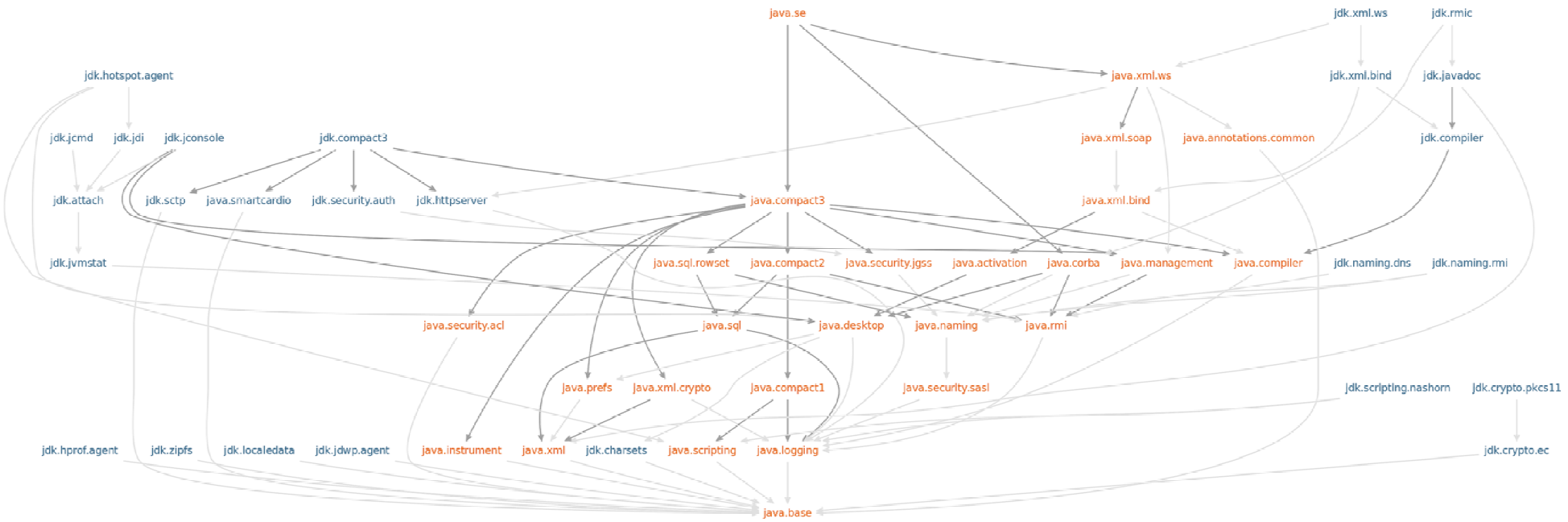
JEP 200: The Modular JDK (3)

Design Principles

- **Standard modules**, whose specifications are governed by the JCP, must have names starting with the string "**java.**".
- **All other modules** are merely part of the JDK, and must have names starting with the string "**jdk.**".
- **If a module exports a type** that contains a public or protected member that, in turn, refers to a type from some other module **then the first module must re-export the public types of the second.** This ensures that method-invocation chaining works in the obvious way.
- Additional principles in JEP 200 text to ensure that code which depends only upon Java SE modules will depend only upon standard Java SE types.

JEP 200: The Modular JDK (4)

Module Graph



Module System Goals

Overall View

- Reliable configuration
  - **replace the brittle, error-prone class-path mechanism with a means for program components to declare explicit dependences upon one another**
- Strong encapsulation
  - **allow a component to declare which of its public types are accessible to other components, and which are not**
  - **module can declare an API to other modules**
  - **packages not on the API are hidden**
- Addressing these goals would enable further benefits:
  - A scalable platform
  - Greater platform integrity
  - Improved performance

Modules

A fundamental new kind of Java component

- **A module is a named, self-describing collection of code & data**
  - **Code is organized as a set of packages containing types**
- It **declares** which other **modules** it ***requires*** in order to be compiled and run
- It **declares** which of its **packages** it ***exports***.
- Module system locates modules
  - **Ensures code in a module can only refer to types in modules upon which it depends**
  - **The access-control mechanisms of the Java language and the Java virtual machine prevent code from accessing types in packages that are not exported by their defining modules.**

Module declarations (1)

A new construct of the Java programming language

- The simplest possible **module declaration** just specifies **the name** of its module:

```
module com.foo.bar { }
```

Module declarations (2)

A new construct of the Java programming language

- **requires** clauses can be added to **declare that the module depends**, by name, **upon some other modules, at both compile time and run time**:

```
module com.foo.bar {
    requires com.foo.baz;
}
```

Module declarations (3)

A new construct of the Java programming language

- **exports** clauses can be added to **declare** that the module makes all, and only, **the public types in some packages available for use by other modules:**

```
module com.foo.bar {
    requires com.foo.baz;
    exports com.foo.bar.alpha;
    exports com.foo.bar.beta;
}
```

  - ***If a module's declaration contains no exports clauses then it will not export any types at all to any other modules.***

Module declarations (4)

A new construct of the Java programming language

- The **source code** for a module declaration is, by convention, placed in a file named **module-info.java** at the root of the module's source-file hierarchy.
- The source files for the com.foo.bar module, *e.g.*, might include:

  module-info.java

  com/foo/bar/alpha/AlphaFactory.java

  com/foo/bar/alpha/Alpha.java

  ...
- A module declaration is compiled, by convention, into a file named module-info.class, placed similarly in the class-file output directory.

Module declarations (5)

A new construct of the Java programming language

- Module names, like package names, must not conflict.
  - **recommended way to name a module is to use the reverse-domain-name pattern**
  - name of a module will often be a prefix of the names of its exported packages
    - but this relationship is not mandatory.
- **A module's declaration does not include a version string,** nor constraints upon **the version strings of the modules** upon which it depends.
  - This is intentional. It is not a goal of the module system to solve that problem.
- Module declarations are part of the Java programming language, rather than a language or notation of their own
  - module information must be available at both compile time and run time

Module Graphs (1)

How do modules relate to each other?

- Running example:
  - An application that uses
    - com.foo.bar module
    - the platform's java.sql module.
  - The module that contains the core of the application is declared as follows:

```
module com.foo.app {
  requires com.foo.bar;
  requires java.sql;
}
```

Module Graphs (2)

Transitive closure computation of dependencies

- **The module system *resolves* the dependences expressed in the app's requires clauses:**

```
do {
    locate additional modules to fulfill those dependences ;
    resolve the dependences of those modules ;
} while (there are dependencies in modules to fulfill) ;
```

- **The result of this transitive-closure computation is a *module graph***
  - **has a directed edge for each module with a dependence fulfilled by another module**

Module Graphs (3)

Transitive closure computation of dependencies

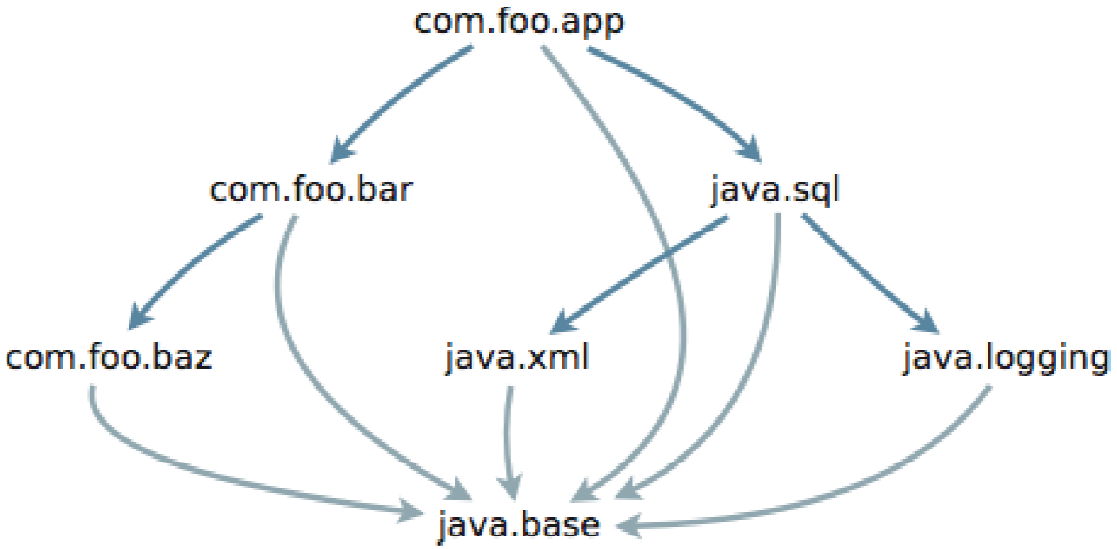
- **To construct a module graph** for the com.foo.app module we need to **inspect the declaration** of the java.sql module:

```
module java.sql {
    requires java.logging;
    requires java.xml;
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
}
```

Module Graphs (4)

Transitive closure computation of dependencies

- **dark blue lines** represent explicit dependence relationships (**requires**)
- **light blue lines** represent the implicit dependences



Module Paths

Where do modules fulfilling dependences come from?

- **module system can select a module to resolve a dependence**
  - built-in to the compile-time or run-time environment or
  - a module defined in an artifact
    - the module system locates artifacts on one or more *module paths* defined by the host system.
- **A module path is a sequence of directories containing module artifacts**
  - searched, in order, for the first artifact that defines a suitable module.
- **Module paths are materially different from class paths, and more robust:**
  - A class path is a means to locate individual types in all the artifacts on the path.
  - A **module path** is a means to **locate whole modules** rather than individual types.
    - If a particular **dependence can not be fulfilled then resolution will fail** with an error message

Accessibility (1)

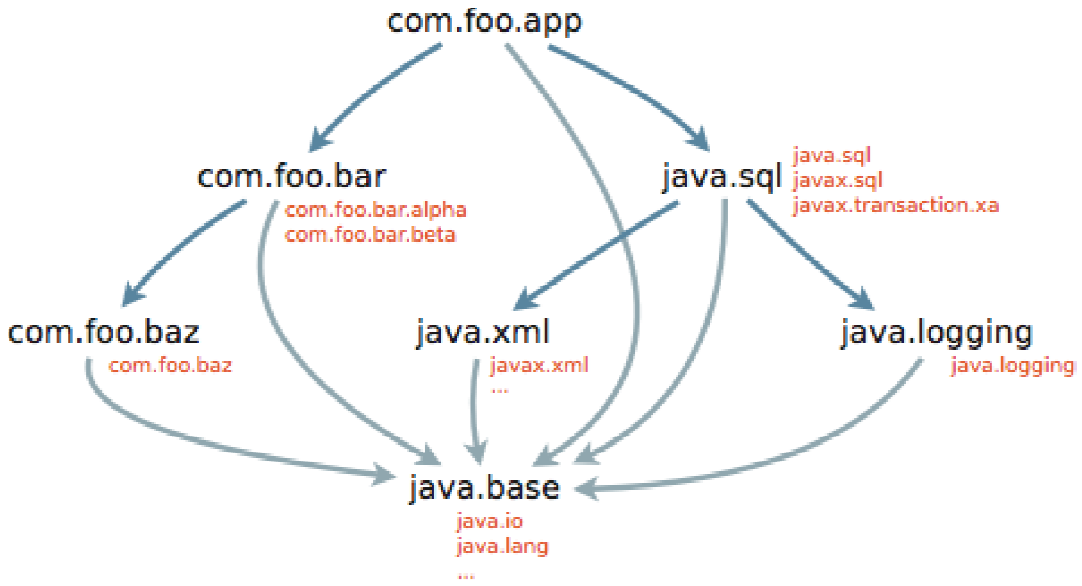
- **The readability relationships defined in a module graph, combined with the exports clauses in module declarations, are the basis of *strong encapsulation*:**
  - **The Java compiler and virtual machine consider the public types in a package in one module to be *accessible* by code in some other module only when**
    - **the first module is readable by the second module, and**
    - **the first module exports that package.**
- That is, if two types S and T are defined in different modules, and T is public, then code in S *can access* T if:
  - S's module reads T's module, and T's module exports T's package.

Accessibility (2)

- **A type referenced across module boundaries that is not accessible in this way is unusable in the same way that a private method or field is unusable:**
  - Any attempt to use it **will cause an error to be reported by the compiler**, or
  - **IllegalAccessError to be thrown by the Java virtual machine**, or
  - **IllegalAccessException to be thrown by reflective run-time APIs.**
- A type declared *public* in a package **not exported** in the declaration of its module will **only be accessible to code in that module**.
- A method or field referenced across module boundaries is accessible if its enclosing type is accessible, and if the declaration of the member itself also allows access.

Accessibility (3)

- To see how strong encapsulation works in the case of the above module graph, we **label each module** with the **packages that it exports**:



Services (1)

Loose coupling

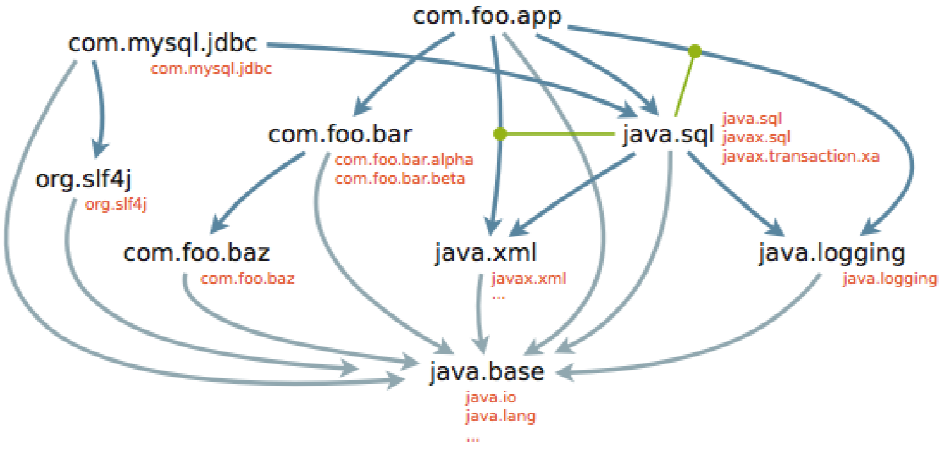
- Our com.foo.app **module extended** to use a MySQL database
  - a MySQL JDBC driver implementing java.sql.Driver is provided in a module:

```
module com.mysql.jdbc {
        requires java.sql;
        requires org.slf4j;
        exports com.mysql.jdbc;
}
```

Services (2)

Loose coupling

- In order for the java.sql module to make use of this driver we must
  - **add the driver module to the run-time module graph**
  - **resolve its dependences**
- ***java.util.ServiceLoader* class can instantiate the driver class via reflection**



Services (3)

Loose coupling

- Module system must be able to locate service providers.
- **Services provided are declared** with a **provides** clause:

```
module com.mysql.jdbc {
    requires java.sql;
    requires org.slf4j;
    exports com.mysql.jdbc;
    provides java.sql.Driver with com.mysql.jdbc.Driver;
}
```

Services (4)

Loose coupling

- Module system must be able to locate service users.
- Services **used** are declared with a **uses** clause:

```
module java.sql {
    requires public java.logging;
    requires public java.xml;
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
    uses java.sql.Driver;
}
```

Services (5)

Advantages of using module declations to declare service relationships

- Clarity
- **Service declarations can be interpreted at compile time**
  - to ensure that the **service interface is accessible**
  - to ensure that providers actually **do implement their declared service interfaces**
  - to ensure that **observable providers are appropriately compiled and linked prior to run time**
- Catching runtime problems at compile time!

Reflection

Inspecting and manipulating the module graph at runtime

- new package **java.lang.module**
- new class **java.lang.reflect.Module** : a single module at run time

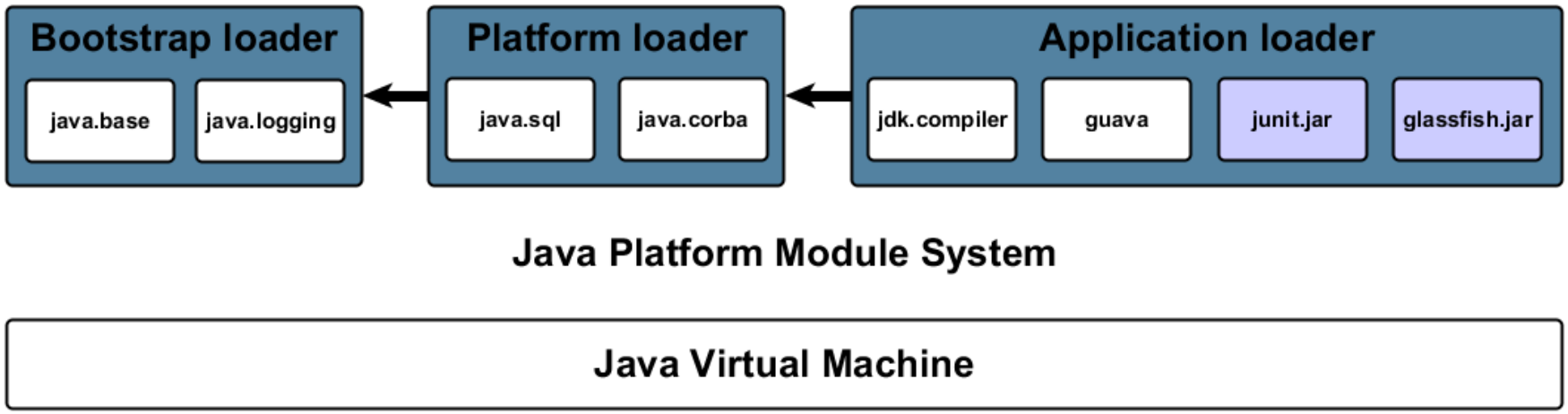
```
public final class Module {
        public String getName();
        public ModuleDescriptor getDescriptor();
        public ClassLoader getClassLoader();
        public boolean canRead(Module source);
        public boolean isExported(String packageName); }
```

- New java.lang.Class::getModule() method.

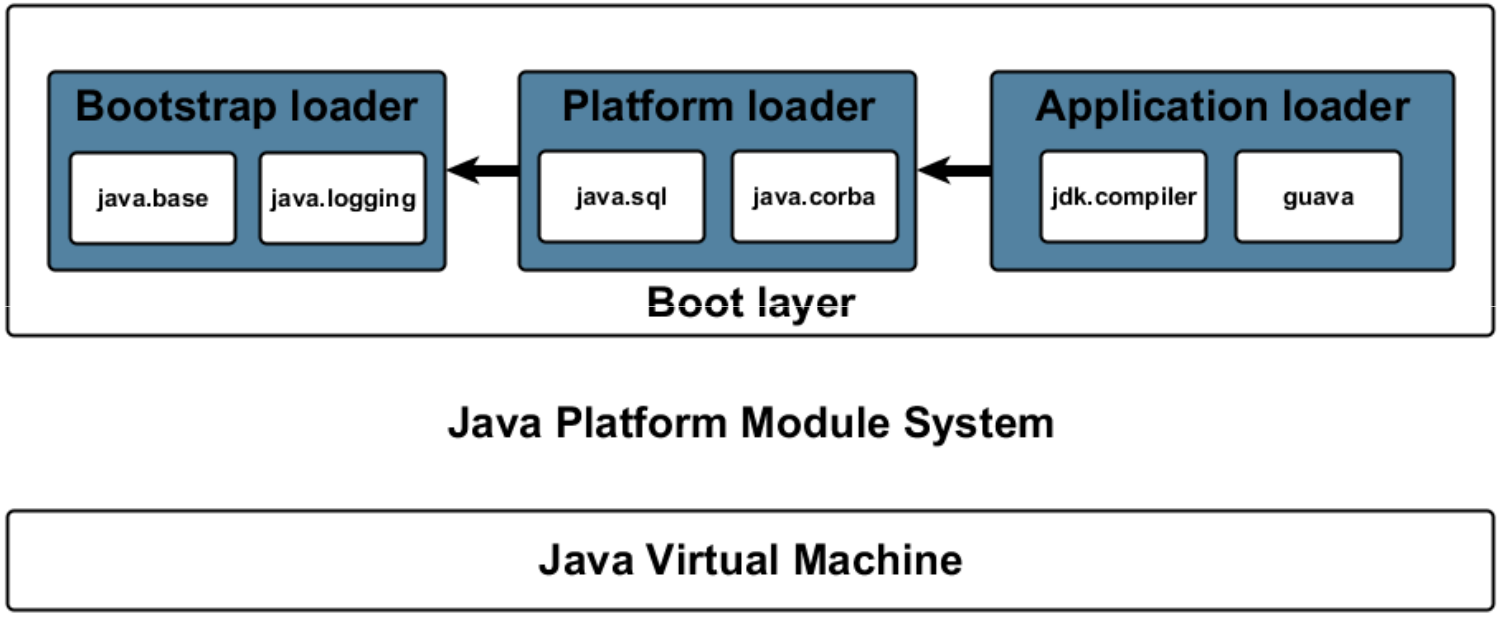
Class Loaders

- **Few restrictions on the relationships between modules and class loaders:**
  - **A class loader can load types from one module or from many modules**
    - **as long the modules do not interfere with each other and**
    - **the types in any particular module are loaded by just one loader**
- **Critical to compatibility**
  - **retains the existing hierarchy of built-in class loaders.**
- Easier to modularize existing applications with complex class loaders
  - **class loaders can be upgraded to load types in modules**
  - **without necessarily changing their delegation patterns**

Modular Class Loading in JDK 9

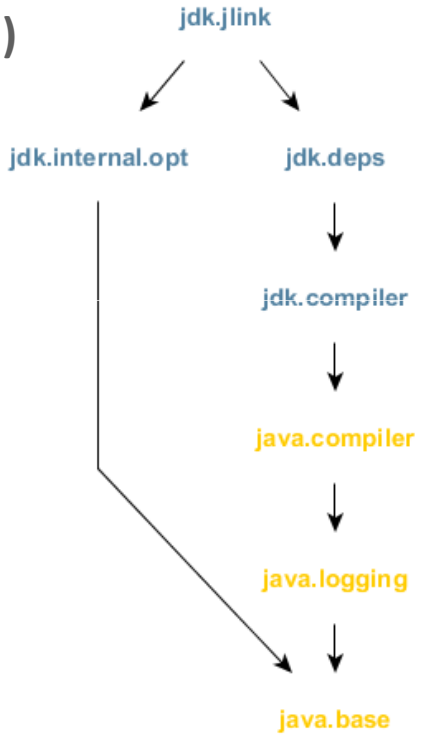


Layers



Layer creation

(1)



(2)

```
String moduleName -> {
   switch (moduleName) {
           case "java.base":
           case "java.logging":
                   return BOOTSTRAP_LDR;
           default:
                   return APP_LDR;
   }
}
```

Unnamed Modules

Backwards compatibility: Loading types from the class path

- **Every class loader has a unique *unnamed module***
  - returned by the new ClassLoader::getUnnamedModule method
- **A class loader loads a type not defined in a named module**
  - that type is considered to be in the unnamed module
- **An unnamed module**
  - **reads every other module**
  - **exports all of its packages to every other module**
- Existing class-path applications using only standard APIs can keep working

sun.misc.Unsafe

- **sun.misc.Unsafe** is an internal and unsafe abstraction in the JDK
  - for building better, faster and safer abstractions in the JDK
  - **Inside:** Java reflection, Java serialization, NIO, java.util.concurrent, java.lang.invoke, CORBA performance, Crypto performance, JMX load average, Mac OS Objective C bridge
  - **Outside:** Akka, Cassandra, Ehcache, Grails, Guava, Hbase, Hadoop, Hazelcast, Hibernate, Jruby, Kafka, Mockito, Neo4j, Netty, Scala, Spark, Spring, others
- **sun.misc.Unsafe is being frozen in Java 9 but still accessible**
  - Cloned internally within the JDK, where internal unsafe features will inevitably increase
- A small set of supported use cases in Java 9
- Significant set of supported use cases anticipated with
- Projects Valhalla and Panama

Unsafe business as usual

- Projects Valhalla and Panama will significantly expand “close to the metal” support in the JDK
- New unsafe abstractions will inevitably appear for use in privileged JDK code
- As those unsafe abstractions mature new safer abstractions will be built on top

Während der Übergangszeit bleibt diese Hintertür offen

At your own risk

- --add-exports <module>/<package>=<target-module>(,<target-module>)* updates <module> to export <package> to <target-module>, regardless of module declaration. <target-module> can be ALL-UNNAMED to export to all unnamed modules.

Jigsaw und die Werkzeuge

- jimage
- jdeps
- jlink

jimage – Modulverzeichnis-Kommando

Tools should never read jimage files, directly or via code. It's an JVM-internal format ..

```
C:\projects\jdk-9> java -version
java version "9-ea"
Java(TM) SE Runtime Environment (build 9-ea+142-jigsaw-nightly-h5677-20161102)
Java HotSpot(TM) Client VM (build 9-ea+142-jigsaw-nightly-h5677-20161102, mixed mode)


C:\projects\jdk-9\lib> jimage list modules


C:\projects\jdk-9\lib> jimage extract modules --dir C:\projects\jdk-9\mydir


C:\projects\jdk-9> java --list-modules
```

jdeps - Java-Class-Dependency-Analyzer

```
C:\projects\jdk-9\mydir> dir
31.05.2016  11:23    <DIR>          java.activation
31.05.2016  11:23    <DIR>          java.annotations.common
31.05.2016  11:23    <DIR>          java.base
31.05.2016  11:23    <DIR>          java.compact1
31.05.2016  11:23    <DIR>          java.compact2
31.05.2016  11:23    <DIR>          java.compact3
31.05.2016  11:23    <DIR>          java.compiler

C:\projects\jdk-9\mydir\java.desktop> jdeps -s module-info.class
module-info.class -> java.base
module-info.class -> java.datatransfer
module-info.class -> java.desktop

C:\projects\jdk-9\mydir\java.desktop> jdeps -v module-info.class
module-info.class -> java.base
module-info.class -> java.datatransfer
module-info.class -> java.desktop
   java.desktop.module-info  -> com.sun.media.sound.WaveFloatFileReader   JDK internal API (java.desktop)
   java.desktop.module-info  -> com.sun.media.sound.WaveFloatFileWriter   JDK internal API (java.desktop)
   java.desktop.module-info  -> java.awt.im.spi.InputMethodDescriptor
   java.desktop.module-info  -> java.net.ContentHandlerFactory
   java.desktop.module-info  -> javax.accessibility.AccessibilityProvider
   java.desktop.module-info  -> javax.imageio.spi.ImageInputStreamSpi
```

Additional diagnostic options supported by the launcher include

java -Xdiag:resolver causes the module system to describe its activities as it constructs the initial module graph

```
C:\projects\jdk-9> java -Xdiag:resolver|more
[Resolver] Root module javafx.web located
[Resolver]   (jrt:/javafx.web)
[Resolver] Root module jdk.xml.dom located
[Resolver]   (jrt:/jdk.xml.dom)
[Resolver] Root module jdk.packager.services located
[Resolver]   (jrt:/jdk.packager.services)
[Resolver] Root module jdk.httpserver located
[Resolver]   (jrt:/jdk.httpserver)
[Resolver] Root module javafx.base located
[Resolver]   (jrt:/javafx.base)
[Resolver] Root module jdk.net located
[Resolver]   (jrt:/jdk.net)
[Resolver] Root module javafx.controls located
[Resolver]   (jrt:/javafx.controls)
[Resolver] Root module java.se located
[Resolver]   (jrt:/java.se)
[Resolver] Root module jdk.compiler located
[Resolver]   (jrt:/jdk.compiler)
[Resolver] Root module jdk.jconsole located
[Resolver]   (jrt:/jdk.jconsole)
```

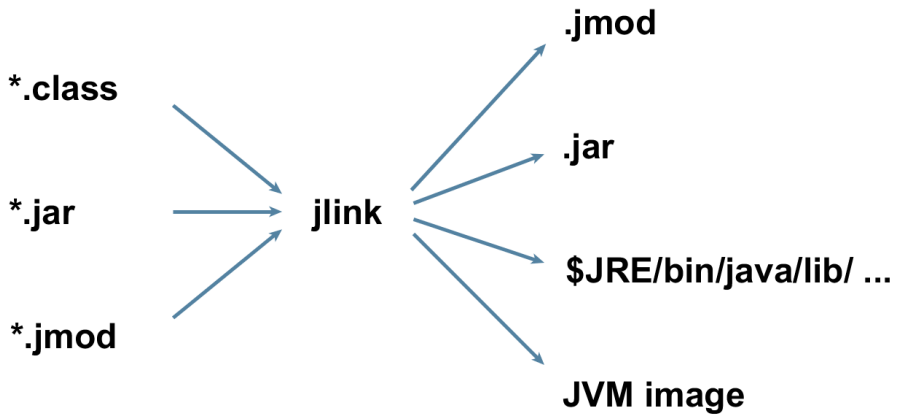
jlink - generiert JRE und Applikations-Images

- Frei wählbares Image-Verzeichnis
- Platzsparende Runtime, inklusive eigener Anwendungsmodule

```
jlink <options> --module-path <modulepath> --output <path>

jlink --module-path $JDKMODS:mlib --add-modules myapp --output myimage

C:\> jlink --module-path c:\projects\jdk-9\jmods;mlib --add-modules com.greetings --output
greetingsapplication
```



jlink - generiert JRE und Applikations-Images (1)

- Image-Verzeichnis C:\greetingsapplication 32,4 MB

```
C:\greetingsapplication> dir

Directory of C:\greetingsapplication
09.03.2017  22:11    <DIR>          .
09.03.2017  22:11    <DIR>          ..
09.03.2017  22:11    <DIR>          bin
09.03.2017  22:11    <DIR>          conf
09.03.2017  22:11    <DIR>          include
09.03.2017  22:11    <DIR>          legal
09.03.2017  22:11    <DIR>          lib
09.03.2017  22:11               166 release
```

- Datei release

```
#Thu Mar 09 22:11:23 CET 2017
OS_NAME="Windows"
MODULES="java.base com.greetings"
OS_VERSION="5.1"
OS_ARCH="i586"
JAVA_VERSION="9"
JAVA_FULL_VERSION="9-ea"
```

jlink - generiert JRE und Applikations-Images (2)

- Image-Verzeichnis C:\greetingsapplication 32,4 MB

```
C:\greetingsapplication\bin> java -m com.greetings/com.greetings.Main
Greetings!

C:\greetingsapplication\bin> dir
09.03.2017  22:11           122.880 java.dll
09.03.2017  22:11           206.336 java.exe
09.03.2017  22:11           206.848 javaw.exe
09.03.2017  22:11            15.360 jimage.dll
09.03.2017  22:11           175.104 jli.dll
09.03.2017  22:11             9.728 keytool.exe
09.03.2017  22:11           455.328 msvcp120.dll
09.03.2017  22:11           970.912 msvcr120.dll
09.03.2017  22:11            75.264 net.dll
09.03.2017  22:11            44.544 nio.dll
09.03.2017  22:11    <DIR>          server
09.03.2017  22:11            34.816 verify.dll
09.03.2017  22:11            62.976 zip.dll

C:\mlib> java -jar com.greetings.jar
Greetings!

C:\mlib> jdeps -v com.greetings.jar
```

jlink - generiert JRE und Applikations-Images (3)

- Image-Verzeichnis C:\greetingsapplication 32,4 MB

```
C:\greetingsapplication\bin> java -m com.greetings/com.greetings.Main
Greetings!


C:\greetingsapplication\bin> java -Xdiag:resolver -m com.greetings/com.greetings.Main
[Resolver] Root module com.greetings located
[Resolver]   (jrt:/com.greetings)
[Resolver] Module java.base located, required by com.greetings
[Resolver]   (jrt:/java.base)
[Resolver] Result:
[Resolver]   com.greetings
[Resolver]   java.base
Greetings!


C:\greetingsapplication\bin> java --list-modules -m com.greetings/com.greetings.Main
com.greetings
java.base@9-ea
```

JSR 376: Java Platform Module System – Gradle (1)

Integration with developer tools (Maven, Gradle, IDE‘s)

- Sources per JDK
  - service
    - java ..
    - java-8 ..
    - java-9
      - org.gradle.example.service
        - Service9

- Dependencies per JDK

```
sources    {
java8      ..
java9        {
           dependencies {
                  library 'org.apache.httpcomponents:httpclient:4.5.1'
      }
   }
}
```

JSR 376: Java Platform Module System – Gradle (2)

Integration with developer tools (Maven, Gradle, IDE‘s)

- **JEP 238: Multi-Release JAR**

```
jar root
    – A.class
    – B.class
    – C.class
    – D.class
    – META-INF
        – versions
            – 8
                – A.class
                – B.class
            – 9
                – A.class
```

Participation

- Download and test JDK 9 Early Access (EA) builds
  - See https://jdk9.java.net/ for downloads
  - If you're a core developer on a FOSS Project, join Quality Outreach effort
    - https://wiki.openjdk.java.net/display/Adoption/Quality+Outreach
    - Let us know about regressions & showstoppers you find testing your project against EA builds
- Provide feedback on Draft JEPs on their OpenJDK mailing lists
  - See http://openjdk.java.net/jeps/0 for list and details
- If you want to contribute changes, see http://openjdk.java.net/contribute/

We want your feedback please

- Read the JEP‘s and other documents linked off Project Jigsaw web site
  - http://openjdk.java.net/projects/jigsaw/
- Subscribe to jigsaw-dev mailing list in OpenJDK
  - Discuss experiences using Project Jigsaw
- Try out Project Jigsaw EA builds available at http://jdk.java.net/jigsaw
- Prepare your code for JDK 9!

Zusammenfassung

- **Die Modularisierung der Java SE Plattform im JDK 9 bringt viele Vorteile, aber auch größere Änderungen**
- Existierender Anwendungs-Code, der nur offizielle Java SE Plattform-API‘s mit den unterstützten JDK-spezifischen API’s verwendet, soll auch weiterhin ohne Änderungen ablauffähig sein
  - Abwärtskompatibilität
  - Dennoch ist es wahrscheinlich, wenn weiterhin veraltete Funktionalität oder JDK-interne API’s verwendet werden, dass der Code unverträglich sein kann
- Entwickler sollten sich frühzeitig damit vertraut machen, wie existierende Bibliotheken & Anwendungen auf JDK 9 anzupassen sind, sie modularisiert werden, welche Designfragen zu klären sind und wie man vorhandenen Anwendungs-Code trotzdem mit JDK 9 zum Laufen bekommt, auch wenn man diesen nicht verändern kann
- **Bitte JDK 9 Early Access Builds ausgiebig testen und Feedback geben**
  - **jigsaw-dev@openjdk.java.net**

Danke!

Wolfgang.Weigend@oracle.com

Twitter: @wolflook