Java SE 9 and the Application Server

Kevin Sutter – MicroProfile and Java EE Architect @kwsutter
Java SE 9
Standalone
Java 9 – Standard Features

JSR 379: Java SE 9 Release Contents
General Announcement from Oracle on Sept 21, 2017
Enhancements Targeted to Java 9

| JEP 201: Modular Source Code         | JEP 231: Remove Launch-Time JRE Version Selection |
| JEP 265: Marlin Graphics Renderer   | JEP 212: Resolve Lint and Doclint Warnings       |
| JEP 102: Process API Updates       | JEP 243: Java-Level JVM Compiler Interface      |
| JEP 258: HarfBuzz Font-Layout Engine| JEP 223: New Version-String Scheme              |
| JEP 143: Improve Contended Locking | JEP 220: Modular Run-Time Images                |
| JEP 257: Update JavaFX/Media to Newer Version of GStreamer | JEP 262: TIFF Image I/O |
| JEP 211: Elide Deprecation Warnings on Import Statements | JEP 263: HiDPI Graphics on Windows and Linux |
| JEP 245: Validate JVM Command-Line Flag Arguments | JEP 266: More Concurrency Updates |
| JEP 244: TLS Application-Layer Protocol Negotiation Extension | JEP 158: Unified JVM Logging |
| JEP 219: Datagram Transport Layer Security (DTLS) | JEP 248: Make G1 the Default Garbage Collector |
| JEP 267: Unicode 8.0                | JEP 249: OCSP Stapling for TLS                   |
| JEP 226: UTF-8 Property Files      | JEP 251: Multi-Resolution Images                |
| JEP 254: Compact Strings           | JEP 276: Dynamic Linking of Language-Defined Object Models |
| JEP 241: Remove the jhat Tool      | JEP 259: Stack-Walking API                      |
| JEP 240: Remove the JVM TI hprof Agent | JEP 274: Enhanced Method Handles                |
| JEP 250: Store Interned Strings in CDS Archives | JEP 271: Unified GC Logging                  |
| JEP 214: Remove GC Combinations Deprecated in JDK 8 | JEP 270: Reserved Stack Areas for Critical Sections |
| JEP 228: Add More Diagnostic Commands | JEP 253: Prepare JavaFX UI Controls & CSS APIs for Modularization |
| JEP 165: Compiler Control          | JEP 221: Simplified Doclet API                  |
| JEP 229: Create PKCS12 Keystores by Default | JEP 225: Javadoc Search |
| JEP 232: Improve Secure Application Performance | JEP 280: Indify String Concatenation |
| JEP 246: Leverage CPU Instructions for GHASH and RSA | JEP 279: Improve Test-Failure Troubleshooting |
| JEP 224: HTML5 Javadoc             | JEP 233: Generate Run-Time Compiler Tests Automatically |
| JEP 215: Tiered Attribution for javac | JEP 230: Microbenchmark Suite                  |
| JEP 217: Annotations Pipeline 2.0 | JEP 110: HTTP/2 Client                         |
| JEP 222: jshell: The Java Shell (Read-Eval-Print Loop) | JEP 272: Platform-Specific Desktop Features |
| JEP 213: Milling Project Coin      | JEP 193: Variable Handles                      |
| JEP 199: Smart Java Compilation, Phase Two | JEP 281: HotSpot C++ Unit-Test Framework       |
| JEP 268: XML Catalogs              | JEP 273: DRBG-Based SecureRandom Implementations|
| JEP 255: Merge Selected Xerces 2.11.0 Updates into JAXP | JEP 238: Multi-Release JAR Files |
| JEP 237: Linux/AArch64 Port         | JEP 269: Convenience Factory Methods for Collections |
| JEP 256: BeanInfo Annotations      | JEP 278: Additional Tests for Humongous Objects in G1 |
| JEP 236: Parser API for Nashorn     | JEP 282: jlink: The Java Linker                |
| JEP 252: Use CLDR Locale Data by Default | JEP 260: Encapsulate Most Internal APIs |
| JEP 264: Platform Logging API and Service | JEP 200: The Modular JDK |
| JEP 197: Segmented Code Cache      | JEP 261: Module System                         |
| JEP 247: Compile for Older Platform Versions | JEP 277: Enhanced Deprecation |
| JEP 216: Process Import Statements Correctly | JEP 283: Enable GTK 3 on Linux |
| JEP 235: Test Class-File Attributes Generated by javac | JEP 275: Modular Java Application Packaging |
| JEP 243: Java-Level JVM Compiler Interface | JEP 223: New Version-String Scheme |
| JEP 220: Modular Run-Time Images | JEP 262: TIFF Image I/O |
| JEP 263: HiDPI Graphics on Windows and Linux | JEP 266: More Concurrency Updates |
| JEP 158: Unified JVM Logging       | JEP 248: Make G1 the Default Garbage Collector |
| JEP 249: OCSP Stapling for TLS      | JEP 251: Multi-Resolution Images                |
| JEP 276: Dynamic Linking of Language-Defined Object Models | JEP 259: Stack-Walking API |
| JEP 274: Enhanced Method Handles    | JEP 271: Unified GC Logging                    |
| JEP 270: Reserved Stack Areas for Critical Sections | JEP 253: Prepare JavaFX UI Controls & CSS APIs for Modularization |
| JEP 221: Simplified Doclet API      | JEP 225: Javadoc Search                        |
| JEP 280: Indify String Concatenation | JEP 279: Improve Test-Failure Troubleshooting |
| JEP 233: Generate Run-Time Compiler Tests Automatically | JEP 230: Microbenchmark Suite |
| JEP 110: HTTP/2 Client             | JEP 272: Platform-Specific Desktop Features |
| JEP 193: Variable Handles          | JEP 281: HotSpot C++ Unit-Test Framework       |
| JEP 273: DRBG-Based SecureRandom Implementations | JEP 238: Multi-Release JAR Files |
| JEP 269: Convenience Factory Methods for Collections | JEP 278: Additional Tests for Humongous Objects in G1 |
| JEP 282: jlink: The Java Linker    | JEP 260: Encapsulate Most Internal APIs        |
| JEP 200: The Modular JDK           | JEP 261: Module System                         |
| JEP 277: Enhanced Deprecation      | JEP 283: Enable GTK 3 on Linux                |
| JEP 275: Modular Java Application Packaging | JEP 275: Modular Java Application Packaging |
Java 9 Structural Changes

- Yes, there is a lot of new functionality, but I'm not talking about those today.
- Focus on two significant changes coming in Java 9 carrying (potentially) large business impact

**Change 1:** The JDK layout is changing

**Change 2:** The (java) world is going modular
Change 1: The JDK directory layout is changing

<table>
<thead>
<tr>
<th>IBM Java 8 GA</th>
<th>OpenJDK Java 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>bin</td>
</tr>
<tr>
<td>include</td>
<td>conf</td>
</tr>
<tr>
<td>jre</td>
<td>management</td>
</tr>
<tr>
<td>linux</td>
<td>security</td>
</tr>
<tr>
<td></td>
<td>include</td>
</tr>
<tr>
<td>jre</td>
<td>linux</td>
</tr>
<tr>
<td>lib</td>
<td>lib</td>
</tr>
<tr>
<td>applet</td>
<td>modules</td>
</tr>
<tr>
<td>boot</td>
<td>security</td>
</tr>
<tr>
<td>cmm</td>
<td></td>
</tr>
<tr>
<td>deploy</td>
<td></td>
</tr>
<tr>
<td>endorsed</td>
<td></td>
</tr>
<tr>
<td>ext</td>
<td></td>
</tr>
<tr>
<td>fonts</td>
<td></td>
</tr>
<tr>
<td>management</td>
<td></td>
</tr>
<tr>
<td>oblique-fonts</td>
<td></td>
</tr>
<tr>
<td>security</td>
<td></td>
</tr>
<tr>
<td>plugin</td>
<td></td>
</tr>
<tr>
<td>lib</td>
<td></td>
</tr>
<tr>
<td>properties</td>
<td></td>
</tr>
<tr>
<td>version</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Old World (pre-JDK9)</th>
<th>New World (JDK9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installs are JRE or JDK.</td>
<td>There is now a single Java install that may contain development tools, or not.</td>
</tr>
<tr>
<td>JDKs contain a JRE, e.g. there are two bin directories.</td>
<td></td>
</tr>
<tr>
<td>Placing code in certain directories conveys rights, i.e. endorsed, and ext dirs.</td>
<td>Endorsed updates become “upgradeable modules” Extensions concept is abandoned.</td>
</tr>
<tr>
<td>Implementation is provided in various (well-known) JAR files such as rt.jar and tools.jar</td>
<td>No more JARs. IDEs etc will need to learn new file formats and locations.</td>
</tr>
<tr>
<td>Supporting paraphernalia such as timezone info and fonts can be seen</td>
<td>Implementations are hidden in modules</td>
</tr>
</tbody>
</table>

Impact: If you rely on the file layout, your application may break!

• IBM will be fixing all IBM products which have any dependency on layout

Recommendation: JDK9 is available

• Start experimenting as soon as feasible...
Change 2: The World is Going Modular

- There have been numerous attempts to bring modularity to Java SE

  - **JSR 277 – Java Module System** *(2005/2006)*
    - Full module system with version support, repository, and SE/EE integration

  - **JSR 291 – Dynamic Component Support for Java SE** *(2006/2007)*
    - Standardize use of OSGi as a modularity system for Java

  - **JSR 294 - Improved Modularity Support in the Java Programming Language** *(2006/2007)*
    - Extending the Java language with “super-packages”

  - **OpenJDK Jigsaw project delivery in Java 7 / Java 8** *(2008 / 2014)*
    - Attempts to bring together the modularity interests into OpenJDK
    - Rebooted in 2014 with the creation of modularity JEPs targeted to Java 9
    - Advanced prototype work was performed in “Jake” repository and regularly pushed to JDK 9

    `hg clone http://hg.openjdk.java.net/jigsaw/jake/`

    `http://mail.openjdk.java.net/mailman/listinfo/jigsaw-dev`
Java Platform Module System – high level goals

- **Reliable configuration of programs**
  - replace the class-path with a new mechanism allowing program components to declare explicit *dependencies* upon one another.

- **Strong encapsulation**
  - enable a component to declare which of its public types are *accessible* to other components, and which are not.
Reliable Configuration

`java -classpath ...` is a linear search through class files, directories, and JARs

```
Begin Here
```

```
java VM
  rt
  jce
  jsse
  plugin
  sunjce_prov.
  dnsns
  ...
  marketing
  workeffort
  ebay
  minerva
  minilang
  accounting
  gutapp
  party
  assetmaint
  htfacility
  pos.
  content
  manufact.
  product
  bi
  workflow
  ecommerce
  oagls
  ...
  googlebase
  order
  ofbiz
  common
  catalina
  base
  datafile
  entity
  widget
  ...
  rome
  jpos18
  jcl
  barcode4j
  freemarker
  serializer
  naming
  jython
  resolver
  mail
  jenks
  jakarta
  log4j
  httpunit
  mx4j
  batik
  fop
  tomcat
  pol
  lucene
  jdom
  commons
  derby
  axis
  ezmorph
  servlets
  jetty
  looks
  jdbm
  bsh
  bsh
  velocity
  vs-commons
  xerces
  xmlapis
  xmlrpc
  xmlgraphics
  ClassNotFoundException
```

"JAR Hell" or "ClassPath Hell" → trying to satisfy competing requirements by simple path ordering

e.g. my app depends upon `foo-v2.jar` and `bar-v2.jar` but `foo-v2.jar` depends upon `bar-v1.jar`

- `classpath foo-v2.jar; bar-v2.jar; bar-v1.jar` → my app “wins”
- `classpath foo-v2.jar; bar-v1.jar; bar-v2.jar` → foo “wins”
Strong Encapsulation: Internal APIs are now hidden

- There are “Java Platform SE APIs”, and then there are those “useful”, internal public types...
  - Often applications will over-reach the APIs into implementation types

- Java 9 will not discriminate!
  - New module boundaries will only export public APIs to strangers

- Your code may be clean, but how about your dependencies?
  - Some types have migrated into API in later releases, e.g. Base64Encoder

- Java 8 (and 9) contains a tool called jdeps that can show you the impact today

```
jdeps -jdkinternals myapplication.jar
```

Warning: JDK internal APIs are unsupported and private to JDK implementation that are subject to be removed or changed incompatibly and could break your application. Please modify your code to eliminate dependency on any JDK internal APIs. For the most recent update on JDK internal API replacements, please check: https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

Recommendation: Run jdeps early to assess the impact of modularity
Basics of the Java Platform Module System

- Modules are defined by a new type of program component
  - The runtime recognizes classes, interfaces, packages, and now modules

- There are updates to the Java language and JVM specifications
  - Introducing new keywords to declare modules, and new behaviors for VM accessibility

- Modules are named in a familiar convention, e.g. com.example.app
  - Modules names starting “java.” provide platform APIs
  - Module names beginning “jdk.” are supporting code

- Practically, modules are created in a source file called module-info.java
  - Placed in the module root directory (ala package-info.java)
  - Compiled to module-info.class by javac
  - Not designed to be extensible by end users (e.g. no annotations)
  - Expected to be recognized by a wide variety of tools to provide and read the module definition
Simple module declaration

/**
 * java.base defines and exports the core
 * APIs of the Java SE platform.
 */

module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
    exports java.net.spi;
    exports java.nio;
    ...
}
Simple module dependency declaration

/**
 * java.base defines and exports the core APIs of the Java SE platform.
 */

module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
    exports java.net.spi;
    exports java.nio;
    ...
}

module com.example.app {
    requires java.base;
    exports com.example.utils;
    exports com.example.stuff;
}

Applications are defined by a module path configuration

- A well-formed configuration of JSR 376 modules comprises a directed graph of module dependencies, rooted at java.base.
  - All modules ultimately depend upon java.base
  - (it is an implied requires if not made explicit)

- These configurations are specified at runtime by a module path
  java -mp <dir1>;<dir2>;<dir3> -m com.example.app

- The responsibility of assembling a coherent and consistent set of modules in the module path is delegated to external tools
  - e.g. Maven, Ivy, Gradle

- In particular, JSR376 modules do not require version information
  - The “works-with” semantics are open to interpretation by configuration tools
  - A missing dependency, or different modules with the same name is an error
Modular JAR files

- JSR376 module information can co-exist with MANIFEST metadata

```
$ jar tf com.example.app-1.0.jar
META-INF/
META-INF/MANIFEST.MF
module-info.class
com/example/utils/MyUtil.class
com/example/stuff/MyStuff.class
com/example/internal/MyInternals.class
```

- JAR files can be both JSR376 modules and OSGi bundles simultaneously
- There is some overlap in the metadata that needs synchronizing
Reflection and Accessibility

- Reflection APIs are enhanced by JSR 376 to allow for testing and adding new module links at runtime
  - You can 'export' a package to another module
    
    ```java
    MyClass.getClass().getModule().addExports(<packageName>, <targetModule>)
    ```
  - Or add a 'reads' relationship to another module
    
    ```java
    MyClass.getClass().getModule().addReads(<sourceModule>)
    ```

- Cannot use reflection to increase accessibility
  - Types are only accessible if the dependency is readable, and the dependency exports the public type.
  - The Java compiler and virtual machine enforce accessibility in the same way as a private method or field, i.e. an attempt to use an inaccessible type will cause
    - an error to be reported by the compiler, or
    - an `IllegalAccessError` to be thrown by the Java virtual machine, or
    - an `IllegalAccessException` to be thrown by reflective run-time APIs.

- But there are runtime options to break out of modularity and ensure your application continues to work.
Rapid innovation in Java

Java releases accelerating

New Java versioning scheme
• \( <yy.m> \) e.g. 18.3 and 18.9

New Java SE Platform every 6 months
Introduction of LTS (Long Term Support) releases
• Every 3 years
• Starting with Java 18.9
• LTS plan is for 3+5 years support

• Java 9 has a 6 month support lifecycle!

** Java 6 EOL Dec 2018
Access to Java technology

IBM SDK for Java 8 SR5
Based upon OpenJ9
Supported until at least 2025

OpenJDK with OpenJ9

Learn more here:
https://adoptopenjdk.net/?variant=openjdk9-openj9
Java SE 9
in a Java EE Environment
Java 9 + WebSphere Roadmap

Phase 1: Toleration
- Java SE 9 development
- WebSphere toleration of Java 9

Java SE 9 release

Phase 2: Exploitation
- WebSphere exploitation of Java 9
- Application exploitation of Java 9
Phase 1: Toleration of Java 9

- Applications using only Java spec interfaces should still work, unchanged.
- Applications that depend on JDK internal APIs or deep reflection may encounter problems.
- Applications depending on third-party software which depend on JDK internal APIs are also vulnerable.
- Open Liberty is “mostly” tolerant of Java 9. Progress is being reported out in GitHub.
- Liberty will initially run in the “unnamed module” space.
  - As a result, applications must also run in the “unnamed module” space.

- IBM JDK migration guide can be found at: [ibm.biz/java9-compat](http://ibm.biz/java9-compat)
- WebSphere analysis of Java 9 can be found at: [ibm.biz/java9-liberty](http://ibm.biz/java9-liberty)
Phase 1: Toleration of Java 9 (cont)

- --illegal-access=permit
  - Default option for Java 9
    - Removes immediate need for the --add-reads, --add-exports, and --add-opens options
  - --illegal-access=warn
  - --illegal-access=debug
  - --illegal-access=deny (default in some future release)

- First time warning message:
  WARNING: Illegal reflective access by $PERPETRATOR to $VICTIM
Phase 2: Java EE Exploitation of Java 9

• It is still unclear how Java modules will interact in the Java EE space
  • We expect there will be Java EE mandated named modules

• An example module-info.java for a Java EE application might look like:

```java
module com.foo.myapp {
  requires javax.servlet;
}
```

• Since Liberty (and OpenLiberty) are OSGi-based runtimes, both OSGi and JPMS modular systems will be maintained
  • JPMS and OSGi applications will both be usable
Phase 2: Java EE Exploitation of Java 9

JPMS modules
OSGi bundles

Boot layer
java.base
java.logging
java.xml

com.myapp
javax.servlet

Source: ibm.biz/java9-osgi
Phase 2: Java EE Exploitation of Java 9

Source: ibm.biz/java9-osgi
Module Dependencies
Module Dependencies

App Layer
- com.myapp

Liberty Layer
- Unnamed Module
- Unnamed Module

Boot layer
- java.base
- java.logging
- java.xml
Bottom-up vs. Top-down migration to JPMS

• **Bottom-up** migration is the most obvious adoption path for JPMS
  1. App Servers adopt JPMS
  2. Third-party software adopts JPMS
  3. Applications adopt JPMS

• **Top-down** migration is also possible with automatic modules
  1. Applications adopt JPMS
  2. App Servers and third-party software used as automatic modules

• In a Java EE environment, top-down migration offers no compelling modularity benefits
  • Simply let Java EE apps run in the “unnamed module” space
  • Wait for software down the stack to adopt JPMS first, then convert apps to use JPMS
Open Liberty

A lightweight open source server runtime ideal for building Java™ microservices and cloud-native apps

- Easy to consume
- Deploy on any cloud for Java™
- Seamlessly transition to WebSphere

https://openliberty.io/
Summary

• Open Liberty mostly tolerates Java 9 already

• Official JPMS + Java EE adoption will not happen for a while.
  • Wait for bottom-up migration
  • Run apps as unnamed modules for now

• Full JPMS adoption will result in smaller deployment sizes, because of reduced JDK size
Thank you!