Java SE Platform Update

Mark Reinhold (@mreinhold)
Chief Architect, Java Platform Group
Oracle

JCP EC Meeting
2015/01/14
Java SE 8 (JSR 337): Key Features

• Annotations on Java Types (JSR 308)
• Date & Time API (JSR 310)
• Lambda Expressions & Streams (JSR 335)
• Compact Profiles
Java SE 8: Additional Features

- **Language**
  - Access to Parameter Names at Runtime (JSR 269 MR, 337)
  - Add Javadoc to javax.tools (JSR 199 MR)
  - Annotations on Java Types (JSR 308)
  - Generalized Target-Type Inference (JSR 335)
  - Lambda Expressions & Virtual Extension Methods (JSR 269 MR, 335)
  - Repeating Annotations (JSR 269 MR, 337)

- **Core Libraries**
  - Base64 Encoding & Decoding
  - Bulk Data Operations for Collections (JSR 335)
  - Concurrency Updates
  - Date & Time API (JSR 310)
  - Enhance Core Libraries with Lambda (JSR 335)
  - JDBC 4.2 (JSR 114 MR, 221 MR)
  - Parallel Array Sorting
  - Statically-Linked JNI Libraries

- **I18n**
  - BCP 47 Locale Matching
  - Unicode 6.2

- **Networking**
  - HTTP URL Permissions

- **Security**
  - Configurable Secure Random-Number Generation
  - Enhance the Certificate Revocation-Checking API
  - Limited doPrivileged
  - NSA Suite B Cryptographic Algorithms
  - TLS Server Name Indication (SNI) Extension

- **XML**
  - Restrict Fetching of External XML Resources (JSR 206 MR)

- **Platform**
  - Compact Profiles (JSR 3 MR, 160 MR, 337)
  - Prepare for Modularization (JSR 173 MR, 206 MR, 337)
Java SE 8: Reception
Java SE 8: Reception
Java SE 8: Adoption

• GitHub project builds on Java SE 8 (Travis CI)
  – August 2014: 2,380
  – January 2015: 4,982

• Many prominent projects will require SE 8 in near-future releases
  – Scala 2.12, Lucene 5.0, Hippo CMS 7.10, OpenNMS 1.14, Weld 3.0, ...

• Typesafe survey (September 2014): Adoption faster than expected
  – A fifth of SE 8 adopters are already running it in production
  – A third are planning to deploy it shortly
Java SE 8: Maintenance Review (closes 2015/02/09) (details)

• Language & VM specification corrections
  – Update the class and interface initialization algorithm
  – Clarify overload resolution with respect to lambda expressions

• Debugging: Support static and default methods in JDI, JDWP, and JDB
  – Not finished in time for SE 8 Final Release

• Deprecation, in preparation for modularization
  – Extension Mechanism
  – Endorsed-Standards Override Mechanism
Java SE 9 (JSR TBD)

• Key features
  – Java Platform Module System (JSR 376)
  – Java SE Platform Modularization (JEPs 200, 220)

• Additional potential features (so far)
  – Small language cleanups (JEPs 211, 213)
  – Enhanced volatiles (JEP 193)
  – Java Read-Eval-Print Loop (REPL) (JEP 222)
  – Datagram Transport Layer Security (JEP 219)
  – HTTP 2 Client API (JEP 110)
Java Platform Module System (JSR 376)

“Approachable, yet scalable”

• Key features:
  – Reliable configuration
  – Secure encapsulation

• ... which enable:
  – Escape from “JAR hell”
  – A modular, scalable platform
    • Both Java SE and, eventually, Java EE
  – Improved platform security and integrity
  – Improved performance
Java Platform Module System: Technology Overview

• **Module** = A named, self-describing collection of code and data
  – Declares which modules are *required* in order to compile and run it
  – Declares which of its API packages are *exported* for use by other modules, and which are not
  – Declares which service interfaces it *uses*, and which services it *provides*

• **Run-time system responsibilities**
  – Locate all required modules, given an initial module (*resolution*)
  – Ensure that modules do not interfere with each other
    • *E.g.*, in case two modules contain packages of the same name
  – Load code on behalf of the compiler and VM
  – Provide a means for other module systems (*e.g.*, OSGi) to locate and resolve modules

• **Language/VM responsibilities**
  – Prevent code from accessing types in non-exported packages
Java SE Platform Modularization (JEPs 200, 220)

• Goals
  – Divide the Java SE Platform into a set of medium-grained modules
    • More flexible than current Compact Profiles
    • Not so fine-grained as to be difficult to use or costly to implement
  – Clearly distinguish between standard SE vs. implementation-specific modules

• Constraints
  – Compatibility
    • An SE 8 application that uses only standard, non-deprecated SE 8 APIs must continue to work, unchanged, on SE 9
  – Performance
    • Static footprint, startup time, memory footprint, and run-time performance of typical implementations must not degrade
      – Except that footprint may increase to accommodate new features
package java.sql;

public interface Connection {
    PreparedStatement prepareStatement(String sql)
            throws SQLException
    {
        ...
    }
}
package java.sql;

public interface Connection {
    PreparedStatement prepareStatement(String sql)
        throws SQLException
    {
        ...
package java.sql;

public interface Connection {

    PreparedStatement prepareStatement(String sql)
    throws SQLException
    {
        ...
    }
}

java.lang not exported by java.base
package java.sql;

public interface Connection {
    PreparedStatement prepareStatement(String sql)
        throws SQLException
    {
        ...
    }
}
package java.sql;

public interface Connection {
    PreparedStatement prepareStatement(String sql)
        throws SQLException
    {
        ...
    }
}

java.sql not exported by java.base
package java.sql;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;
public class DriverManager {
  ...
}
package java.sql;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;
public class DriverManager {
...

sun.reflect not exported by java.base
package java.sql;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;
public class DriverManager {
...

sun.reflect not exported by java.base
package java.sql;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;
public class DriverManager {
...

sun.reflect not exported by java.base
```java
package java.sql;
import sun.reflect.CallerSensitive;
import sun.reflect.Reflection;
public class DriverManager {
    ...
}
```

**sun.reflect not exported by java.base**
package java.sql;

import java.util.logging.Logger;

public class Driver {
    public Logger getParentLogger() throws SQLFeatureNotSupportedException {
    }
}
package com.foo.app;
import java.sql.*;
...

Driver d = DriverManager.getDriver(uri);
Connection c = d.connect(uri, props);
d.getParentLogger().info("Connection acquired");
package com.foo.app;
import java.sql.*;
...

Driver d = DriverManager.getDriver(uri);
Connection c = d.connect(uri, props);
d.getParentLogger().info("Connection acquired");
package com.foo.app;
import java.sql.*;
...

Driver d = DriverManager.getDriver(uri);
Connection c = d.connect(uri, props);
d.getParentLogger().info("Connection acquired");
The diagram illustrates the relationship between different file extensions and the `jlink` tool in Java.

- **`.class`**: Linked to `jlink`, indicating that it is a class file.
- **`.jar`**: Linked to `jlink`, indicating that it is a Java archive.
- **`.jmod`**: Linked to `jlink`, indicating that it is a JMOD file.
- **`.jmod`**: Linked to `$JRE/bin/java /lib/...`, indicating that it is used as part of the JVM image.
- **`.jar`**: Linked to `$JRE/bin/java /lib/...`, indicating that it is used as part of the JVM image.

The `jlink` tool is used to create a JVM image that includes all the necessary class files and resources.
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.
Java SE Platform Update

Mark Reinhold (@mreinhold)
Chief Architect, Java Platform Group
Oracle

JCP EC Meeting
2015/01/14