You, me and jigsaw

Tom Schindl <tom.schindl@bestsolution.at>

Twitter: @tomsontom
Blog: http://tomsondev.bestsolution.at
Website: http://www.bestsolution.at
About Tom

- CTO BestSolution.at Systemhaus GmbH
- Eclipse Committer
  - e4
  - Platform
  - EMF
- Project lead
  - e(fx)clipse
- Twitter: @tomsontom
- Blog: tomsondev.bestsolution.at
- Corporate: http://bestsolution.at
The target of jigsaw
The target of jigsaw

- Main target split of the rt.jar into smaller pieces
The target of jigsaw

- Main target split of the rt.jar into smaller pieces
  - Easier to maintain
The target of jigsaw

- Main target split of the rt.jar into smaller pieces
  - Easier to maintain
  - Avoid people from accessing internal APIs (misc.Unfafe)
The target of jigsaw

- Main target split of the rt.jar into smaller pieces
  - Easier to maintain
  - Avoid people from accessing internal APIs (misc.Unsafe)
  - Allow to ship stripped down JREs with applications
The target of jigsaw

- Main target split of the rt.jar into smaller pieces
- Easier to maintain
- Avoid people from accessing internal APIs (misc.Unsafe)
- Allow to ship stripped down JREs with applications

```bash
user@system: java -listmods
java.activation@9-ea
java.annotations.common@9-ea
java.base@9-ea
java.compiler@9-ea
java.compiler2@9-ea
java.compiler3@9-ea
// ...
```
The target of jigsaw
The target of jigsaw

- The 2nd target is to allow applications developers to write modular applications
The target of jigsaw

- The 2nd target is to allow applications developers to write modular applications
  - Using the Java9 modules for your code is OPTIONAL
The target of jigsaw

- The 2nd target is to allow applications developers to write modular applications
  - Using the Java9 modules for your code is OPTIONAL
  - Java code not included in a module is wrapped at runtime in an UNAMED-Module
The target of jigsaw

- The 2nd target is to allow applications developers to write modular applications
  - Using the Java9 modules for your code is OPTIONAL
  - Java code not included in a module is wrapped at runtime in an UNAMED-Module
  - Your existing application (whether OSGi based or not) will run unmodified on Java9
The target of jigsaw

- The 2nd target is to allow applications developers to write modular applications
  - Using the Java9 modules for your code is OPTIONAL
  - Java code not included in a module is wrapped at runtime in an UNAMED-Module
  - Your existing application (whether OSGi based or not) will run unmodified on Java9
  - (as long as you not touch into JDK internals)
Show some source code

- You can find out information at runtime using `Class#getModule()`
Show some source code

- You can find out information at runtime using `Class#getModule()`

```java
public class Sample1 {
    public static void main(String[] args) {
        System.err.println( String.class
            + " is in Module "
            + String.class.getModule().getName() + "" );
        System.err.println( javax.swing.JPanel.class
            + " is in Module "
            + javax.swing.JPanel.class.getModule().getName() + "" );
        System.err.println( Sample1.class
            + " is in Module "
            + Sample1.class.getModule().getName() + "" );
    }
}
```
Show some source code

You can find out information at runtime using Class#getModule()

```java
public class Sample1 {
    public static void main(String[] args) {
        System.err.println( String.class
            + " is in Module "
            + String.class.getModule().getName() + "" );
        System.err.println( javax.swing.JPanel.class
            + " is in Module "
            + javax.swing.JPanel.class.getModule().getName() + "" );
        System.err.println( Sample1.class
            + " is in Module "
            + Sample1.class.getModule().getName() + "" );
    }
}
```

```
javac src/sample/Sample1.java
java -cp src sample.Sample1
```
Show some source code

You can find out information at runtime using `Class#getModule()`

```java
public class Sample1 {
    public static void main(String[] args) {
        System.err.println( String.class
                + " is in Module "
                + String.class.getModule().getName() + "" );
        System.err.println( javax.swing.JPanel.class
                + " is in Module "
                + javax.swing.JPanel.class.getModule().getName() + "" );
        System.err.println( Sample1.class
                + " is in Module "
                + Sample1.class.getModule().getName() + "" );
    }
}
```

`javac src/sample/Sample1.java`
`java -cp src sample.Sample1`

```
user@system: java sample.Sample1
class java.lang.String is in Module 'java.base'
class javax.swing.JPanel is in Module 'java.desktop'
class sample.Sample1 is in Module 'null'
```

(c) BestSolution.at - Licensed under Creative Commons Attribution-NonCommerical-ShareAlike 3.0
Anatomy of a Java9 Module

- Module information is encoded in a java file named module-info.java/.class
Anatomy of a Java 9 Module

- Module information is encoded in a java file named module-info.java/.class

OSGi
MANIFEST.MF

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: myfoobar
Require-Bundle: com.foo,
 com.bar
Export-Package: com.mybar,
 com.myfoo
```
Anantomy of a Java9 Module

- Module information is encoded in a java file named module-info.java/.class

**OSGi MANIFEST.MF**

- Manifest-Version: 1.0
- Bundle-ManifestVersion: 2
- Bundle-SymbolicName: myfoobar
- Require-Bundle: com.foo,
  com.bar
- Export-Package: com.mybar,
  com.myfoo

**Java9 module module-info.java**

```java
module myfoobar{
    requires com.foo;
    requires com.bar;
    exports com.mybar;
    exports com.myfoo;
}
```
Anantomy of a Java9 Module

- Things to take a way
  - Java9 modules use the OSGi-Require-Bundle strategy to express dependencies
  - Java9 modules explicitly have to export packages like OSGi to make them accessible
Show some source code

- Build a module
Show some source code

‣ Build a module

```java
module BasicModule {
}
```

d module-info.java
Show some source code

» Build a module

```java
module BasicModule {
}
```

**module-info.java**

```
javac -modulepath ../mods -d ../mods/BasicModule src/module-info.java src/sample/Sample1.java
touch -modulepath ../mods -m BasicModule/sample.Sample1
```
Show some source code

- Build a module

```java
module BasicModule {
}
```

**module-info.java**

```bash
javac -modulepath ../mods -d ../mods/BasicModule src/module-info.java src/sample/Sample1.java
java -modulepath ../mods -m BasicModule/sample.Sample1
```

```
src/sample/Sample1.java:9: error: package javax.swing does not exist
    System.err.println( javax.swing.JPanel.class
   ^
src/sample/Sample1.java:11: error: package javax.swing does not exist
   + javax.swing.JPanel.class.getModule().getName() +"" );
   ^
2 errors
class java.lang.String is in Module 'java.base'
Exception in thread "main" java.lang.IllegalArgumentException: class sample.Sample1 (in module BasicModule) cannot access class javax.swing.JPanel (in module java.desktop) because module BasicModule does not read module java.desktop
    at sample.Sample1.main(BasicModule/Sample1.java:9)
```
Show some source code

- Build a module

```java
module BasicModule {
}
```
Anantomy of a Java9 Module

‣ Is it really based on Require-Bundle????
  ‣ Yep it is

‣ Is that a problem
  ‣ Yep IMHO it is
The Require-Bundle problem

pre Java9 world
The Require-Bundle problem

pre Java9 world

- guava-19.jar
- lib-a.jar
- lib-b.jar
The Require-Bundle problem

pre Java9 world

Diagram:

- guava-19.jar
- lib-a.jar
- lib-b.jar
The Require-Bundle problem

pre Java9 world
The Require-Bundle problem

Java9 world
The Require-Bundle problem

Java9 world

lib-a.jmod  →  MyApp.jmod  ←  lib-b.jmod
The Require-Bundle problem

Java9 world

*google.guava-19.jmod*

*lib-a.jmod*

*lib-b.jmod*

*myApp.jmod*
The Require-Bundle problem

Java9 world

- google.guava-19.jmod
- com.google.guava.guava-19.jmod
- lib-a.jmod
- lib-b.jmod
- myApp.jmod
Java9 Module and Reflection

- OSGi modularity is built on Classloaders
- Java9 modularity is built on the VM
- Suppose the following situation:

  - Module A
    - export bar
  - Module B
    - export foo
  - Module C
    - requires ModuleA & ModuleB
Java9 Module and Reflection
Java9 Module and Reflection

```java
import serializer.Serializer;
import sample.Sample1;

public class User {
    public static void main(String[] args) {
        System.err.println("VALUE: " + Serializer.createInstance(Sample1.class));
    }
}
```
Java9 Module and Reflection

```java
import serializer.Serializer;
import sample.Sample1;

public class User {
    public static void main(String[] args) {
        System.err.println("VALUE: " + Serializer.createInstance(Sample1.class));
    }
}

public class Serializer {

    public static <T> T createInstance(Class<T> clazz) {
        try {
            java.lang.reflect.Constructor<? extends T> c = clazz.getDeclaredConstructors()[0];
            c.setAccessible(true);
            return (T)c.newInstance();
        } catch (Throwable t) {
            throw new RuntimeException(t);
        }
    }
}
```
Java9 Module and Reflection

```java
import serializer.Serializer;
import sample.Sample1;

public class User {
    public static void main(String[] args) {
        System.err.println("VALUE: " + Serializer.createInstance(Sample1.class));
    }
}

public class Serializer {

    public static <T> T createInstance(Class<T> clazz) {
        try {
            java.lang.reflect.Constructor<?> c = clazz.getDeclaredConstructors()[0];
            c.setAccessible(true);
            return (T)c.newInstance();
        } catch (Throwable t) {
            throw new RuntimeException(t);
        }
    }
}

Produces a runtime exception
```
import serializer.Serializer;
import sample.Sample1;

public class User {
    public static void main(String[] args) {
        System.err.println("VALUE: " + Serializer.createInstance(Sample1.class));
    }
}

public class Serializer {

    public static <T> T createInstance(Class<T> clazz) {
        try {
            Serializer.class.getModule().addReads(clazz.getModule());
            java.lang.reflect.Constructor<?> c = clazz.getDeclaredConstructors()[0];
            c.setAccessible(true);
            return (T)c.newInstance();
        } catch (Throwable t) {
            throw new RuntimeException(t);
        }
    }
}
Java9 Module and Reflection

```java
import serializer.Serializer;
import Sample.Sample1;

public class User {
    public static void main(String[] args) {
        System.err.println("VALUE: "+ Serializer.createInstance(Sample1.class));
    }
}

public class Serializer {

    public static <T> T createInstance(Class<T> clazz) {
        try {
            Serializer.class.getModule().addReads(clazz.getModule());
            java.lang.reflect.Constructor<? super clazz> c = clazz.getDeclaredConstructors()[0];
            c.setAccessible(true);
            return (T) c.newInstance();
        } catch (Throwable t) {
            throw new RuntimeException(t);
        }
    }
}
```
Having to add read edges dynamically just to enable reflection is painful, and could slow migration and adoption. Consider relaxing the access model so that reflection does not require, or perhaps simply assumes, readability.

Proposal: Adopt the second alternative suggested in the summary. Revise the core reflection APIs (java.lang.reflect) to assume that any module that contains code that invokes a reflective operation can read the module that defines the types that are the subject of that operation.

As a consequence, most code that uses reflection will not have to take the trouble to add readability edges manually. Code of this form that was previously added to the prototype implementation will be removed.

This proposal does weaken the fidelity story a tiny bit, in the sense that you'll be able to do more at run time than you can in earlier phases, but that seems a worthwhile tradeoff in order to ease migration.

- Mark
Final step package up

- Constructing your custom Java-Install is called linking
Final step package up

- Constructing your custom Java-Install is called linking

```
user@host: jlink --modulepath ../..jmods:../mods/ --addmods Serializer,User --output myimage
```
Final step package up

- Constructing your custom Java-Install is called linking

```bash
user@host: jlink --modulepath ../jmods:/mods/ --addmods Serializer,User --output myimage
```

```bash
user@host: myimage/bin/java -listmods
BasicModule
Serializer
User
java.base@9-ea
java.datatransfer@9-ea
java.desktop@9-ea
java.prefs@9-ea
java.xml@9-ea
```
Final step package up

† Constructing your custom Java-Install is called linking

```
user@host: jlink --modulepath ../jmods.../mods/ --addmods Serializer,User --output myimage
```

```
user@host: myimage/bin/java -listmods
BasicModule
Serializer
User
java.base@9-ea
java.datatransfer@9-ea
java.desktop@9-ea
java.prefs@9-ea
java.xml@9-ea
```

```
user@host: myimage/bin/java user.User
```
So what’s a .jmod file?
So what’s a .jmod file?

- jmod is the deployment unit for Java9 modules
So what’s a .jmod file?

- jmod is the deployment unit for Java9 modules
- the format of jmod is NOT fixed
So what’s a .jmod file?

- jmod is the deployment unit for Java9 modules
- the format of jmod is NOT fixed
- what’s guaranteed is that jmods can be read using NIO.2
So what’s a .jmod file?

- jmod is the deployment unit for Java9 modules
- the format of jmod is NOT fixed
- what’s guaranteed is that jmods can be read using NIO.2

```java
class JModSample {
    public static void main(String[] args) throws Exception {
        FileSystem fs = FileSystems.newFileSystem(Paths.get(".../jmods/java.activation.jmod"), null);
        Path root = fs.getPath("/");
        Files.walkFileTree(root, new SimpleFileVisitor<Path>() {
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
                System.err.println("Found: " + file);
                return FileVisitResult.CONTINUE;
            }
        });
    }
}
```
e4 and Java9 Modules
e4 and Java9 Modules

- e4 & OSGi applications run unmodified on Java9
e4 and Java9 Modules

- e4 & OSGi applications run unmodified on Java9

- Could e4 ditch OSGi in favor of Java9 modules
e4 and Java9 Modules

- e4 & OSGi applications run unmodified on Java9
- Could e4 ditch OSGi in favor of Java9 modules
  - Short them no
e4 and Java9 Modules

- e4 & OSGi applications run unmodified on Java9
- Could e4 ditch OSGi in favor of Java9 modules
  - Short them no
  - Midterm probably – we (BestSolution) are looking at that doing that for e4 on JavaFX
e4 and Java9 Modules

- e4 & OSGi applications run unmodified on Java9

- Could e4 ditch OSGi in favor of Java9 modules
  - Short them no

- Midterm probably - we (BestSolution) are looking at doing that for e4 on JavaFX
  - Biggest stumbling block was/is the OSGi-Service-Registry