redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- In jdk9 come modules - project jigsaw.
- This change primarily means, that you will not be able to use internal apis.
- The restriction is both runtime and compile time.

- it's much more then sun.internal
- *jdk.internal.misc.Unsafe*
- have hardcoded exceptions
- (both for and against)

- more about jigsaw in another jdk9 talk

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- I'm no jigsaw expert
- I'm going to share **bad** practices
- but those may be your only practices if you depend on private apis
- although feature complete, jdk9 is still under development

Porting of *sun.internal* to JDK9

- What's new in java command
- java [options] -p <modulepath> -m <modulename>[/<mainclass>] [args...]
  -p <module path>
- --upgrade-module-path <module path>...
  A : separated list of directories, each directory
  is a directory of modules that replace upgradeable
  modules in the runtime image
- -m / --module <modulename>[/<mainclass>]
- --add-modules <modulename>[,<modulename>...]
  root modules to resolve in addition to the initial module.
  <modulename> can also be ALL-DEFAULT, ALL-SYSTEM,
  ALL-MODULE-PATH.
- --limit-modules <modulename>[,<modulename>...]
  limit the universe of observable modules
- --list-modules [<modulename>[,<modulename>...]]
  list the observable modules and exit
- --dry-run create VM but do not execute main method.
  This --dry-run option may be useful for **validating** the
  command-line options such as **the module system configuration.**

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- What's new in javac command
- javac <options> <source files>
- -m,--add-module...upgrade-module-path <module path>...
- --module-source-path <module-source-path>
  Specify where to find input source files for multiple modules
- --processor-module-path <path>
- --release <release>

So with java/javac not much of unexpected, but what about -X?

Porting of *sun.internal* to JDK9

java -X

- -Xcomp forces compilation of methods on first invocation
- -Xmn<size> sets the initial and maximum size (in bytes) of the heap for the young generation (nursery)
- -Xverify sets the mode of the bytecode verifier
- --add-reads <module>=<target-module>(,<target-module>)*
  updates <module> to read <target-module>, regardless of module declaration.
  <target-module> can be ALL-UNNAMED to read all unnamed modules.
- --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.
- --patch-module <module>=<file>(:<file>)*
  Override or augment a module with classes and resources in JAR files or directories.

javac -X

- .....
- -Xmodule:<module>
  Specify a module to which the classes being compiled belong.

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- What they are for?
- lets javac class in
  package java.lang;
- */usr/lib/jvm/java-1.8.0-openjdk/bin/javac -d . Test.java*
- */usr/lib/jvm/java-1.9.0-openjdk/bin/javac -d . Test.java*

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- compiled by 8
- */usr/lib/jvm/java-1.8.0-openjdk/bin/java java.lang.Test*
- not run
- */usr/lib/jvm/java-1.9.0-openjdk/bin/java java.lang.Test*
- even worse

- Long story short “internal” is there for some time

Porting of *sun.internal* to JDK9

better example:

- package sun.applet;

now

- */usr/lib/jvm/java-1.8.0-openjdk/bin/javac -d . Test.java*

and

- */usr/lib/jvm/java-1.8.0-openjdk/bin/java sun.applet.Test*

works

Porting of *sun.internal* to JDK9

however although

- */usr/lib/jvm/java-1.9.0-openjdk/bin/javac -d . Test.java* works
- */usr/lib/jvm/java-1.9.0-openjdk/bin/java sun.applet.Test* still fails

Porting of *sun.internal* to JDK9

- back to our class in package java.lang;

  /usr/lib/jvm/java-1.9.0-openjdk/bin/javac -d . Test.java

  Test.java:1: error: package exists in another module: java.base
  package java.lang;
  ^

  1 error

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- What module it can be in?
- My class should be part of this module!

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- **jmod** prints out information about the given module
- naive approach
- *for x in `find  /usr/lib/jvm/java-1.9.0-openjdk/jmods -type f` ; do echo xxx `basename $x` xxx ; /usr/lib/jvm/java-1.9.0-openjdk/bin/jmod list $x | grep java.lang ; done*

- java.lang eh... found
  - no longer so easy to move class between packages
  - New importance of package private
- sun.applet ... match!

Porting of *sun.internal* to JDK9

- more “jdk9” approach
- **jdeps** prints *package-> used package-> module*
- */usr/lib/jvm/java-1.9.0-openjdk/bin/jdeps ~/icedtea-web-image/share/icedtea-web/plugin.jar*

- what modules is it part of
- what modules is it using
- what packages possibly moved
- what packages will be internal
- ....

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- Unexpected update:
- **jar** now have switch of *--print-module-descriptor*

  /usr/lib/jvm/java-1.9.0-openjdk/bin/jar --print-module-descriptor --file*=/home/jvanek/icedtea-web-image/share/icedtea-web/plugin.jar*
- As descriptor is compiled special “class”

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

- Even more unexpected update
  - --file=...jar
- beware of no-descriptor jars
- **jar** now can create module
- **jar** can even change jar to jmod

redhat

Porting of *sun.internal* to JDK9

OpenJDK 9

- Lets assume our class **should** belong to java.base

  /usr/lib/jvm/java-1.9.0-openjdk/bin/javac -d . -Xmodule:java.base Test.java

  - Yepi!

Porting of *sun.internal* to JDK9

- Still, how to run it?

/usr/lib/jvm/java-1.9.0-openjdk/bin/java -cp . java.lang.Test
Error: Could not find or load main class java.lang.Test

- Unlike javac, java don't has -Xmodule
- Unlike java -h, -X have some answers
  - --add-reads <module>=<target-module>(,<target-module>)*
    updates <module> to read <target-module>, regardless of module declaration.
    <target-module> can be ALL-UNNAMED to read all unnamed modules.
  - --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.
  - --patch-module <module>=<file>(:<file>)*
    Override or augment a module with classes and resources in JAR files or directories.

Porting of *sun.internal* to JDK9

- There is important naming convention

  ALL-UNNAMED – represents all classes not belonging to any module

- --add-reads <module>=<target-module>(,<target-module>)*
  updates <module> to read <target-module>, regardless of module declaration.
  <target-module> can be ALL-UNNAMED to read all unnamed modules.
- --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.
- --patch-module <module>=<file>(:<file>)*
  Override or augment a module with classes and resources in JAR files or directories.

Porting of *sun.internal* to JDK9

- most powerful is **patch-module**
  - --patch-module <module>=<file>(:<file>)*
- you can add classes to module in start time
- */usr/lib/jvm/java-1.9.0-openjdk/bin/java -cp . java.lang.Test*
  Error: Could not find or load main class java.lang.Test
  x
- */usr/lib/jvm/java-1.9.0-openjdk/bin/java --patch-module **java.base=.** java.lang.Test*
  Hello World

- In this case we added classes in packages in current directory to java.base module.
- java.lang.Test can now access rest of java.base without restrictions
- However it is **hiding** the added resources **inside module**

Porting of *sun.internal* to JDK9

- see this (03) example – two packages, both “internal”, cooperating
- When compiled by 9

  Test2.java:1: error: package exists in another module: java.management

  - See the ALL-UNNAMED, thats getting cryptic, we will get there
- For simplicity compile by 8

  */usr/lib/jvm/java-1.8.0-openjdk/bin/javac -d . *.java*
- Simple sun.applet call to sun.com
- Ok on 8
- Bad on 9

  /usr/lib/jvm/java-1.8.0-openjdk/bin/java -cp . sun.applet.Test1
  Hello World
  /usr/lib/jvm/java-1.9.0-openjdk/bin/java -cp . sun.applet.Test1
  Error: Could not find or load main class sun.applet.Test1
- patching can go too wrong

  /usr/lib/jvm/java-1.9.0-openjdk/bin/java --patch-module java.base=. -cp . sun.applet.Test1

  Error occurred during initialization of VM
  java.lang.reflect.LayerInstantiationException: Package sun.applet in both module java.base and module java.desktop

Porting of *sun.internal* to JDK9

- **<u>One single package can be in one single module</u>**
- There is no way to workaround
- That is valid also for ALL-UNNAMED
- So all packages in your legacy classpath must not collide with packages in modules

Porting of *sun.internal* to JDK9

- Looking closely: --patch-module <module>=<file>(:<file>)*
- There can be more then one
- however
  /usr/lib/jvm/java-1.9.0-openjdk/bin/java --patch-module ***java.desktop=./repacked/m1***
  --patch-module ***java.management=./repacked/m2*** *-cp . sun.applet.Test1*

  Exception in thread "main" java.lang.IllegalAccessError: class sun.applet.Test1 (in module java.desktop) cannot access class javax.management.Test2 (in module java.management) because module java.desktop does not read module java.management

- It's easy to imagine
- You have all those classes in single jar (=>no op, see slide before)
- or the need of both directional accessibility (=> no op without more work)

Porting of *sun.internal* to JDK9

- New classes needs to be made accessible
- --add-reads <module>=<target-module>(,<target-module>)*
  updates <module> to read <target-module>
- --add-exports <module>/<package>=<target-module>(,<target-module>)*
  updates <module> to export <package> to <target-module>,

- Unluckily in **most cases** you need **both**
- Unless the modules already **reads** each other

/usr/lib/jvm/java-1.9.0-openjdk/bin/java --patch-module java.desktop=./repacked/m1 --patch-module java.management=./repacked/m2 ***--add-reads java.management=java.desktop --add-reads java.desktop=java.management*** *-cp . sun.applet.Test1*

Hello World!

- The demo is constructed so you are using your own classes in private packages
- Patched files are exported by default
- However you will face the same when you want to use non exported classes in jdk9's modules
- Only you will **not** need --patch-module, but much **more** --add-exports
- Now adding your non private, normal api using those

Porting of *sun.internal* to JDK9

- Now real using of internal api:
  - my.pkg.Test1 is using something in sun.awt.AppContext and sun.awt.SunToolkit;
    - Compile by jdk8 – ok, but 4 warnings of:
    - “internal proprietary API and may be removed in a future release”
  - Compile by jdk9
    - Try a guess :)
    - 4 warnings => 4 errors
  - Jdk class, so nothing to patch into java.desktop
  - Luckily the runtime error is much better
    - *Exception in thread "main" java.lang.IllegalAccessError: class my.pkg.Test1 (in unnamed module @0x2ac1fdc4) cannot access class sun.awt.AppContext (in module java.desktop) because module java.desktop does not export sun.awt to unnamed module*
- /usr/lib/jvm/java-1.9.0-openjdk/bin/java **--add-exports java.desktop/sun.awt=ALL-UNNAMED** my.pkg.Test1
  - And here it goes

Porting of *sun.internal* to JDK9

- You can imagine that complex private-api dependent application can get into interesting problems
  - Hopefully ITW will remain the lonely one
    - http://icedtea.classpath.org/hg/icedtea-web/rev/7e15398e117d
    - http://icedtea.classpath.org/hg/icedtea-web/rev/ef2c2def287a
    - http://icedtea.classpath.org/hg/icedtea-web/rev/2cb12ef65318
    - http://icedtea.classpath.org/hg/icedtea-web/rev/f243a4832f32
      =>
    - http://icedtea.classpath.org/hg/icedtea-web/file/f243a4832f32/launcher/launchers.in
      Or better
    - icedtea-web-image/bin/javaws
- Unluckily…
  - Itw is container
  - Most of the non-jdk9 legacy javaws apps and applets do not work
    - Even fact that **java.version** returns 9 instead of 1.X.whatever **is killing** them…
    - So think about various security exception and access control issues...

Porting of *sun.internal* to JDK9

Conclusion:

- This is **not** the way
- It is way if you really need to run application **compiled** by **jdk8** or older **in jdk9 ecosystem**
- If application is maintained, it is better to **rebuild** as **proper module**, and have **special jdk9** version
- and we have introduced some tools how to look through that maze