



Java 9 Performance

By Jeroen Borgers



Proactive



Contents

- Introduction
- Modular Java
 - Overview & performance
- Compiler improvements & API
- Improved locking
- Variable handles
- Diagnostics
- Garbage collector
- Compact Strings
- Summary and Conclusions
- Questions



Introduction

- Java 8 introduced lambda's and (parallel) streams
- Java 9 introduces Jigsaw
- Will life change with Java 9?
- What about performance?



Schedule

- 2015/12/10 Feature Complete
- 2016/02/04 All Tests Run
- 2016/02/25 Rampdown Start
- 2016/04/21 Zero Bug Bounce
- 2016/06/16 Rampdown Phase 2
- 2016/07/21 Final Release Candidate
- 2016/09/22 General Availability



Schedule

- **2015/12/10** **Feature Complete**
- 2016/02/04 All Tests Run
- 2016/02/25 Rampdown Start
- 2016/04/21 Zero Bug Bounce
- 2016/06/16 Rampdown Phase 2
- 2016/07/21 Final Release Candidate
- **2016/09/22** **General Availability**



Schedule

- **Now:** **EA jigsaw-b86, b90**
- 2015/12/10 Feature Complete
- 2016/02/04 All Tests Run
- 2016/02/25 Rampdown Start
- 2016/04/21 Zero Bug Bounce
- 2016/06/16 Rampdown Phase 2
- 2016/07/21 Final Release Candidate
- 2016/09/22 General Availability



How will life change?

- No more rt.jar, tools.jar in Java runtime
 - Tools like IntelliJ and Eclipse currently rely on it and will not run
 - Modules instead: added logical layer
 - Accessible at runtime via URL:
 - `jrt:/java.base/java/lang/String.class`
- Unrecognized VM options
 - Deprecated in JDK 8, removed now
 - `-XX:MaxPermSize`



How will life change? -2

- Several Java API's not accessible anymore
 - internal, unsupported and not portable: `sun.*`, `com.sun.*`, `java.awt.peer`
 - `jdeps` from Java 8 helps to find static dependencies
- G1 default collector
- `'_'` no longer allowed as identifier by itself
- private interface methods (instance and static) possible
 - To complete default and static interface methods of Java 8
- No more support for `java -source` and `-target < 1.6`



Project Jigsaw goals



Project Jigsaw goals

- Make platform&JDK more easily scalable down to small computing devices;
- Improve security and maintainability
- Enable **improved application performance**; and
- Make it easier for developers to construct and maintain libraries and large applications.



Platform Module System, JSR 376 - Improved performance

- Platform, library, and application components are put in one runtime and dependencies are known
- Ahead-Of-Time and Whole-Program optimizations are more effective



Modules enable optimizations

- Known where code will be used, optimizations more feasible;
- JVM-specific memory images that load faster than class files;
 - Fast lookup of both JDK and application classes;
- early bytecode verification;
- ahead-of-time (AOT) compilation of method bodies to native code;
- the removal of unused fields, methods, and classes; and
- aggressive inlining of, e.g., lambda expressions.

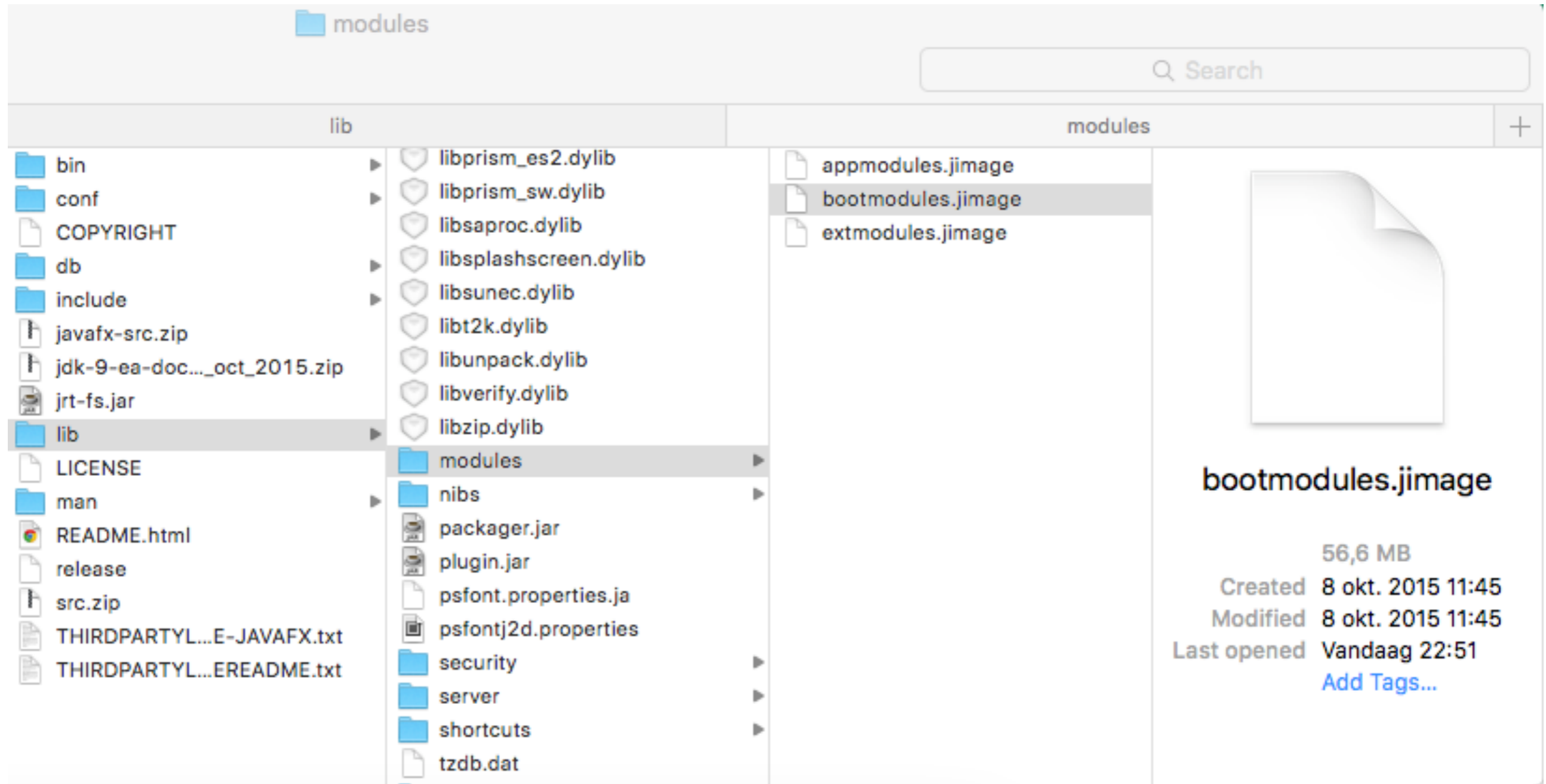


Startup Performance

- Current JVM startup:
 - class loading slow: executes a linear scan of all JARs on classpath
 - Annotation detection requires to read all classes in package(s)
 - Spring: `<context:component-scan base-package="your.package.name" />`
 - Modules will provide a fast class-lookup, including by annotation, without reading all class files
 - Indexes created when the module is compiled



Modular Java - JEP 220: Modular Run-Time Images





inside the .jimage file - jimage tool

```
bin -- -bash -- 97x11
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep java/lang/Object.class
```

- Demo



jimage tool

```
bin -- -bash -- 97x5
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep java/lang/Object.class
/java.base/java/lang/Object.class
Jeroens-MacBook-Pro-2:bin jeroen$
Jeroens-MacBook-Pro-2:bin jeroen$
```



jimage tool

```
bin — -bash — 97x11
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep java/lang/Object.class
/java.base/java/lang/Object.class
Jeroens-MacBook-Pro-2:bin jeroen$
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep /ThreadLocal[^$]*class
```



jimage tool

```
bin -- -bash -- 97x11
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep java/lang/Object.class
/java.base/java/lang/Object.class
Jeroens-MacBook-Pro-2:bin jeroen$
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep /ThreadLocal[^$]*class
/java.base/java/lang/ThreadLocal.class
/java.base/java/util/concurrent/ThreadLocalRandom.class
/java.base/sun/nio/cs/ThreadLocalCoders.class
/java.xml/com/sun/xml/internal/stream/util/ThreadLocalBufferAllocator.class
Jeroens-MacBook-Pro-2:bin jeroen$
```



jimage and jdeps tool

```
bin -- -bash -- 97x11
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep java/lang/Object.class
/java.base/java/lang/Object.class
Jeroens-MacBook-Pro-2:bin jeroen$
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep /ThreadLocal[^$]*class
/java.base/java/lang/ThreadLocal.class
/java.base/java/util/concurrent/ThreadLocalRandom.class
/java.base/sun/nio/cs/ThreadLocalCoders.class
/java.xml/com/sun/xml/internal/stream/util/ThreadLocalBufferAllocator.class
Jeroens-MacBook-Pro-2:bin jeroen$
```

```
bin -- -bash -- 97x10
Jeroens-MacBook-Pro-2:bin jeroen$ jdeps -module java.lang.String
```




















jimage and jdeps tool

```
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep java/lang/Object.class
/java.base/java/lang/Object.class
Jeroens-MacBook-Pro-2:bin jeroen$
Jeroens-MacBook-Pro-2:bin jeroen$ ./jimage list ../lib/modules/bootmodules.jimage | grep /ThreadLocal[^$]*class
/java.base/java/lang/ThreadLocal.class
/java.base/java/util/concurrent/ThreadLocalRandom.class
/java.base/sun/nio/cs/ThreadLocalCoders.class
/java.xml/com/sun/xml/internal/stream/util/ThreadLocalBufferAllocator.class
Jeroens-MacBook-Pro-2:bin jeroen$
```

```
Jeroens-MacBook-Pro-2:bin jeroen$ jdeps -module java.lang.String
java.base -> java.base
  java.lang (java.base)
    -> java.io
    -> java.nio.charset
    -> java.util
    -> java.util.regex
    -> java.util.stream
    -> jdk.internal
    java.base
    java.base
    java.base
    java.base
    java.base
    JDK internal API (java.base)
Jeroens-MacBook-Pro-2:bin jeroen$
```

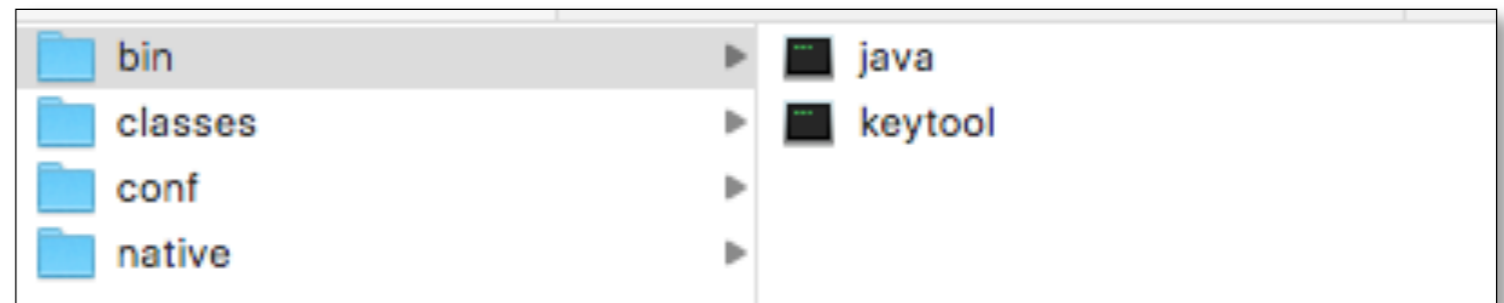
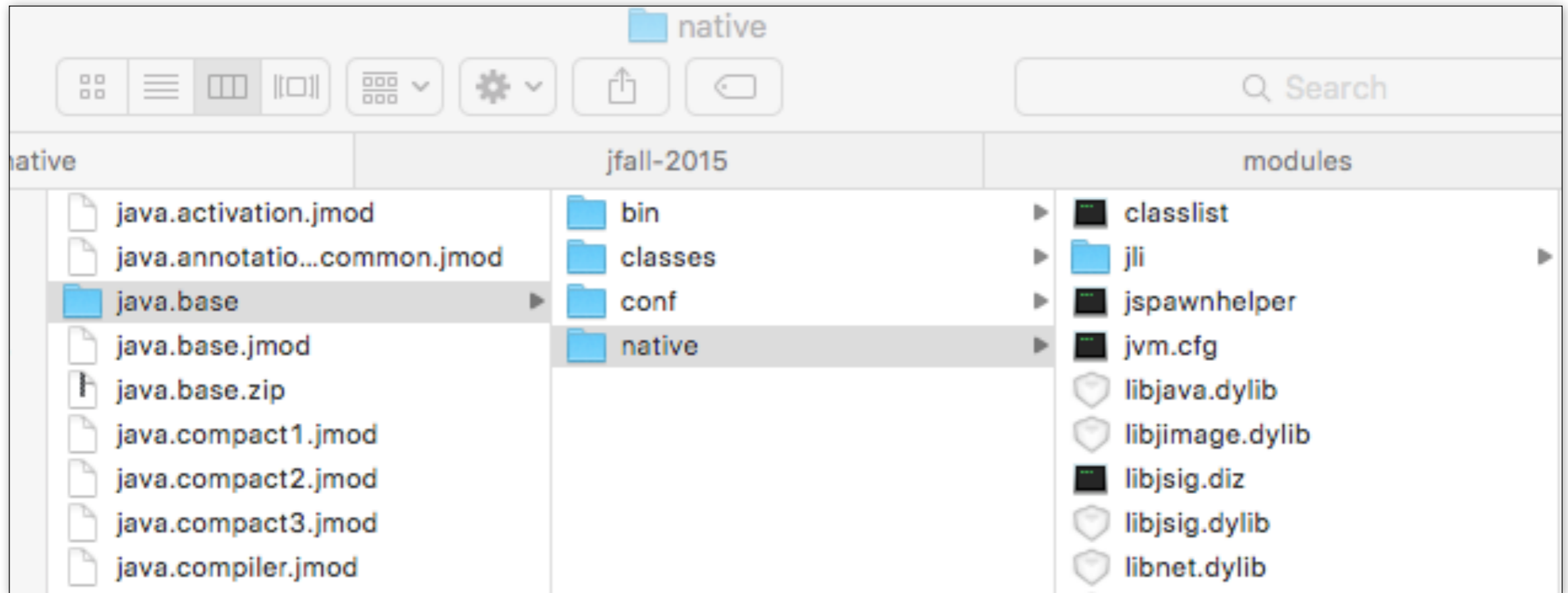


Packaging: JMOD files

jmods	
Name	Size ▾
 java.base.jmod	56 MB
 java.desktop.jmod	13,1 MB
 javafx.web.jmod	11,5 MB
 jdk.localedata.jmod	7,2 MB
 jdk.compiler.jmod	6,1 MB
 javafx.graphics.jmod	5 MB
 java.xml.jmod	4,5 MB
 jdk.deploy.jmod	4,1 MB
 java.xml.ws.jmod	2,7 MB
 jdk.hotspot.agent.jmod	2,6 MB
 javafx.controls.jmod	2,5 MB
 java.corba.jmod	2,5 MB
 jdk.scripting.nashorn.jmod	2,2 MB
 jdk.charsets.jmod	1,8 MB
 jdk.xml.bind.jmod	1,8 MB
 javafx.media.jmod	1,7 MB



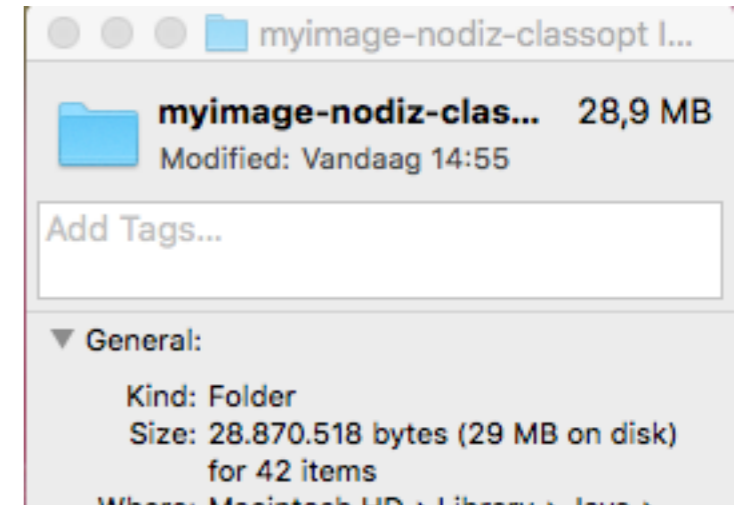
jmod = jar++ for compile and link time





jlink example

```
bin — -bash — 92x5
Jeroens-MacBook-Pro-2:bin jeroen$ ./jlink --modulepath ../jmods --addmods java.base --output
myimage-nodiz-classopt --exclude-files *.diz --class-optimize : all
```



- native debug files excluded
- Small size image: 29 MB, can be 12 MB
- Class optimization plugin
 - Class.forName removal when accessible



Compiler improvements

- JEP 165: Compiler Control
 - method specific flags, file: `inline: ["+java.util.*", "-com.sun.*"]`
 - runtime manageable: `jcmd <pid> Compiler.add_directives <file>`
- JEP 199: Smart Java Compilation
 - `sjavac`: smart wrapper around `javac`
 - incremental compiles - recompile only what's necessary
 - parallel compilation - utilize cores during compilation
 - keep compiler in hot VM - reuse JIT'ed `javac` instance for consecutive invocations



Compiler API - JEP 243

- Allow Java code to observe, query, and affect JVM's compilation
- Pluggable JIT compiler architecture
 - Graal
- May persist code profile and reuse it AOT, avoid JVM warm-up
 - Like Azul's ReadyNow!



JEP 143: Improve contended locking

- 22 many-threads benchmarks
- Field reordering and cache line alignment
- Speed up PlatformEvent::unpark()
- Fast Java monitor enter and exit operations
- Fast Java monitor notify/notifyAll operations



JEP 193: Variable handles

- Typed reference to a variable
- Atomicity for object fields, array elements and ByteBuffers
 - like `java.util.concurrent.atomic`, `sun.misc.Unsafe` operations
 - `java.lang.invoke.VarHandle`, next to `MethodHandle` from Java7
 - `java.util.concurrent` will move from use of `Unsafe` to `VarHandles`
 - VH will use `Unsafe` internally
- What is that `Unsafe` class? In thread stacks I see: `Unsafe.park`

Every time I see this:

**java.lang.Thread.State: WAITING
at sun.misc.Unsafe.park(Native Method)**

I think on this:



By: @arturotena

Unsafe.park - 2





Side step: sun.misc.Unsafe

- Better alternative to native C or assembly code via JNI
- Atomic compare-and-swap operations like in AtomicInteger, ConcurrentHashMap

```
public final native boolean compareAndSwapInt(Object o, long  
offset, int expected, int x)
```

- Direct access to native, off-heap memory

```
public native long allocateMemory(long bytes); //quite unsafe!
```

- Creating objects without calling constructor like in Serialization
- High performance; special handling by JVM
 - methods are intrinsified: assembler instruction inlined to caller, no JNI-call overhead



Side step: `sun.misc.Unsafe`

- Access to `Unsafe` is restricted to JDK classes however
 - Can be worked around by reflection
- Java 9 puts `Unsafe` in `jdk` internal module
 - Safe and updated alternatives come available: `VarHandles`
- Libs currently using `Unsafe`: Netty, Hazelcast, Kryo, Cassandra, Spring, Akka, ..
- command line flag makes `Unsafe` readable for transition period



JEP 193: Variable handles

- Use case:

```
class Position {  
  
    private volatile int x = 0;  
  
    public void walkRight() {  
  
        x++;  
  
    }  
  
}
```

- Is it thread safe?



JEP 193: Variable handles

- Use case:

```
class Position {  
  
    private volatile int x = 0;  
  
    public void walkRight() {  
  
        x++;  
  
    }  
  
}
```

- Not thread-safe because `x++` is in fact two operations:

```
int tmp = this.x;  
  
this.x = tmp + 1;
```

- Other thread may `walkRight` in between these two and have his result lost



JEP 193: Variable handles

- Solution:

```
class Position {  
  
    private AtomicInteger x = new AtomicInteger();  
  
    public void walkRight() {  
  
        x.incrementAndGet();  
  
    }  
  
}
```

- memory usage compared to previous?



JEP 193: Variable handles

```
class Pos {  
  
    private int x = 0;  
  
    public void walkRight() {  
  
        x = VH_POS_X.addAndGet(this, 1);  
  
    }  
  
}
```



JEP 193: Variable handles

```
class Pos {  
    private static final VarHandle VH_POS_X;  
    private int x = 0;  
    static {  
        try {  
            VH_POS_X = MethodHandles.lookup().  
                in(Pos.class).findFieldVarHandle(Pos.class, "x", int.class);  
        } catch (Exception e) { throw new Error(e); }  
    }  
    public void walkRight() {  
        VH_POS_X.addAndGet(this, 1);  
    }  
}
```



More diagnostic commands

```
Jeroens-MacBook-Pro-2:Home jeroen$ jcmd 31142 VM.class_hierarchy
31142:
java.lang.Object/null
|--java.lang.reflect.Proxy$ProxyBuilder$$Lambda$122/123322386/null
|--jdk.internal.jimage.ImageBufferCache/null
|--org.netbeans.core.windows.view.ModeAccessor/0x00007faf026d8730 (intf)
|-java.lang.invoke.LambdaForm$DMH/1841321848/null
```

```
Jeroens-MacBook-Pro-2:Home jeroen$ jcmd 31142 VM.stringtable
```

```
\31142:
StringTable statistics:
Number of buckets      :      60013 =    480104 bytes, avg   8.000
Number of entries      :      17882 =    429168 bytes, avg  24.000
Number of literals     :      17882 =   1604736 bytes, avg  89.740
Total footprint        :              =   2514008 bytes
Average bucket size    :      0.298
Variance of bucket size :      0.299
Std. dev. of bucket size:      0.547
Maximum bucket size    :              4
```

- `Compiler.queue .codelist, .codecache`
- `VM.set_flag`



G1 as default collector

- G1 default on 32 and 64 bit server configs
- Replaces Parallel GC as default
 - Parallel GC shows long pauses for large heaps
- JDK8_u40 / JEP 156: G1 now supports class unloading instead of needing a full GC
- Optimizes for low pause time
 - Not for throughput nor CPU load!
- May need more tuning
 - `-XX:MaxGCPauseMillis=n`



Compact Strings

- Improve space efficiency of String, StringBuilder, etc.
- String is often biggest consumer of the heap
- Characters are UTF-16: 2 bytes, while most apps use only Latin-1: 1 byte
- New: byte[] or char[], + encoding flag field
- Less allocation, less GC, less data on bus: so also better time efficiency!
- SPECjbb2005 server app benchmark:
 - 21% less live data
 - GC: 21% less frequent, 17% less long
 - 10% better app throughput

Java 9 Performance Summary and Conclusions

- Modules
 - Big incompatible change in JDK 9
 - Performance optimizations introduced and enabled
 - class loading, startup time, more aggressive optimizations
 - Internal, fast Unsafe features made available with VarHandles
 - Innovation on compilers front
 - Faster javac, more control, pluggable JIT, AOT
 - Faster dealing with more data and threads
 - G1, compact strings, contention

Java 9 Performance Questions?



Want to learn more?

- www.jpoinpoint.com / www.profactive.com
 - references, presentations
- Accelerating Java Applications
 - 3 days technical training
 - March 2015
 - nl-jug members 10% discount
 - hand-in business card today: 15% discount





Please rate my talk in the official J-Fall app!