Java 9: Tips on Migration and Upgradability

Bernard Traversat
Vice President of Development
Java SE Platform
Oracle

JavaYourNext (Cloud)

November, 2017

Safe Harbor Statement

The following 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.

Agenda

1. Upgradability
2. JDK 9 migration tips
3. Why Java 9 New Module really matters for upgradability ?
4. Why all the above matter even more for the cloud and microservices!

Upgradability

Software Upgradability

It's a very hard problem!

Managing and Securing the Cloud at Scale

- Massive cloud buildup over the past few years means compounding risks if a patch or update is not applied
  - Equifax failed to patch a known security vulnerability
- Through 2020, **99% of vulnerabilities** exploited will continue to be the ones known by security and IT professionals for at least one year (Gartner)
- The 2016 Microsoft Security Intelligence Report states that 5,000 to 6,000 new vulnerabilities surface each year. That works out to an average of **15 per day**.
- Exponential complexity to manage n different versions of the same software (n^m)
- The Cloud is demanding continuous upgradability!

Why Upgrading?

- Security fixes
- Regressions fixes
- New release or a new feature
- More performance
- More robustness
- Because the layer underneath or above you just upgraded (OS's, Hardware)
- Reduce cost of operation via uniformity

Why Not Upgrading?

InformationWeek
THE BUSINESS VALUE OF TECHNOLOGY

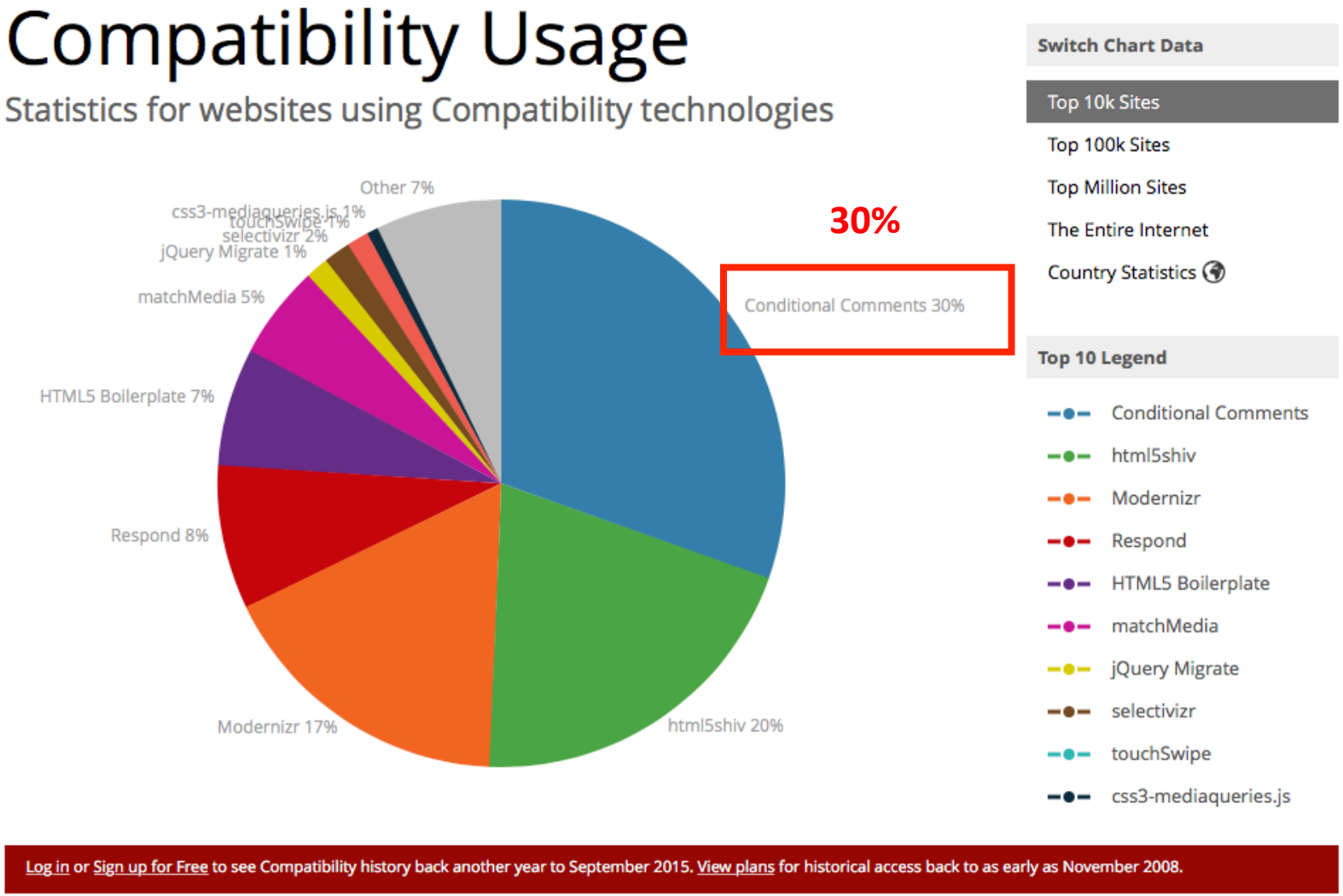
IT Downtime Costs $26.5 Billion In Lost Revenue

Something is going to go wrong!

It is going to cost me a lot of money if something goes wrong!

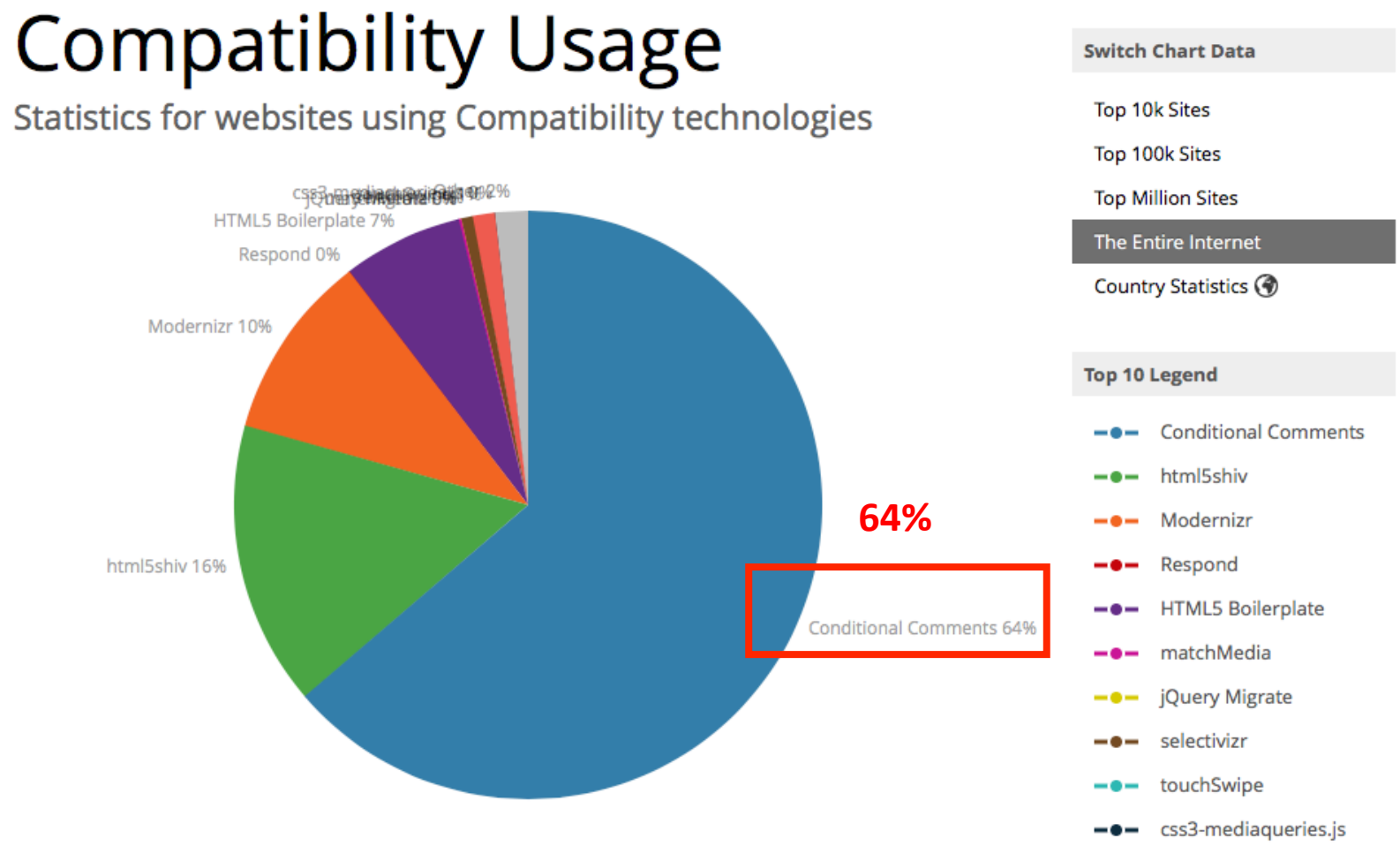
It is work. I need to modify my app!

JavaScript Upgradability



https://trends.builtwith.com/javascript/compatibility

JavaScript Upgradability



https://trends.builtwith.com/javascript/compatibility

Golang Upgradability

Go 1 came with a Compatibility Promise, which gave developers peace of mind that their programs will continue to compile and run correctly as long as the Go 1 specification is in place. Now, the Go 2 specification is to be designed, existing the possibility to break compatibility with 1.x. Cox considers that new features need to be carefully selected because

> Go 2 must bring along all those developers [including those using Go 1.x]. We must ask them to unlearn old habits and learn new ones only when the reward is great.
>
> Go 2 must also bring along all the existing Go 1 source code. We must not split the Go ecosystem. Mixed programs, in which packages written in Go 2 import packages written in Go 1 and vice versa, must work effortlessly during a transition period of multiple years. We'll have to figure out exactly how to do that; automated tooling like go fix will certainly play a part.

To avoid disruption Google will limit the number of new features to "two or three, certainly not more than five," and "each change will require careful thought, planning, and tooling." Such features do not include minor changes such as "allowing identifiers in more spoken languages or adding binary integer literals," because "they are easier to get right." Cox talked about major changes such as "additional support for error handling, or introducing immutable or read-only values, or adding some form of generics, or other important topics not yet suggested."

Python Upgradability

Surgery is good, but painful!

Is Python 3.0 backward-compatible and why?

Python 3.0 implements a lot of very useful features and breaks backward compatibility. It does it on purpose, so the great features can be implemented even despite the fact Python 2.x code may not work correctly under Python 3.x.

So, basically, **Python 3.0 is not backward-compatible on purpose**. **Thanks to that, you can benefit from a whole new set of features.** It is even called "*Python 3000*" or "*Python 3K*".

From "*What's new in Python 3.0*" (available here):

> Python 3.0, compared to 2.6. Python 3.0, also known as "Python 3000" or "Py3K", **is the first ever intentionally backwards incompatible Python release**. There are more changes than in a typical release, and more that are important for all Python users. Nevertheless, after digesting the changes, you'll find that Python really hasn't changed all that much – by and large, **we're mostly fixing well-known annoyances and warts, and removing a lot of old cruft**.

C# and .Net Upgradability

Microsoft's open source fork of the .NET platform, called .NET Core, will be modified for better compatibility with existing applications, says Program Manager Immo Landwerth in a recent post.

When the company embarked on its .NET Core effort, it took a minimalist approach, stripping out legacy code in order to get the best performance and smallest footprint for cross-platform server applications.

The consequence is that porting existing applications is hard, because so many of the existing .NET Framework APIs are missing.

Java Upgradability

There are three kinds of compatibility explicitly managed in the Java Platform:

- Binary: will existing libraries and applications still link?
- Source: will existing source bases still compile? If the source bases still compile, will the resulting class file have equivalent semantics?
- Behavioral: at runtime, will existing libraries and applications behave in a sufficiently similar way?

Categories of APIs in the Java Platform

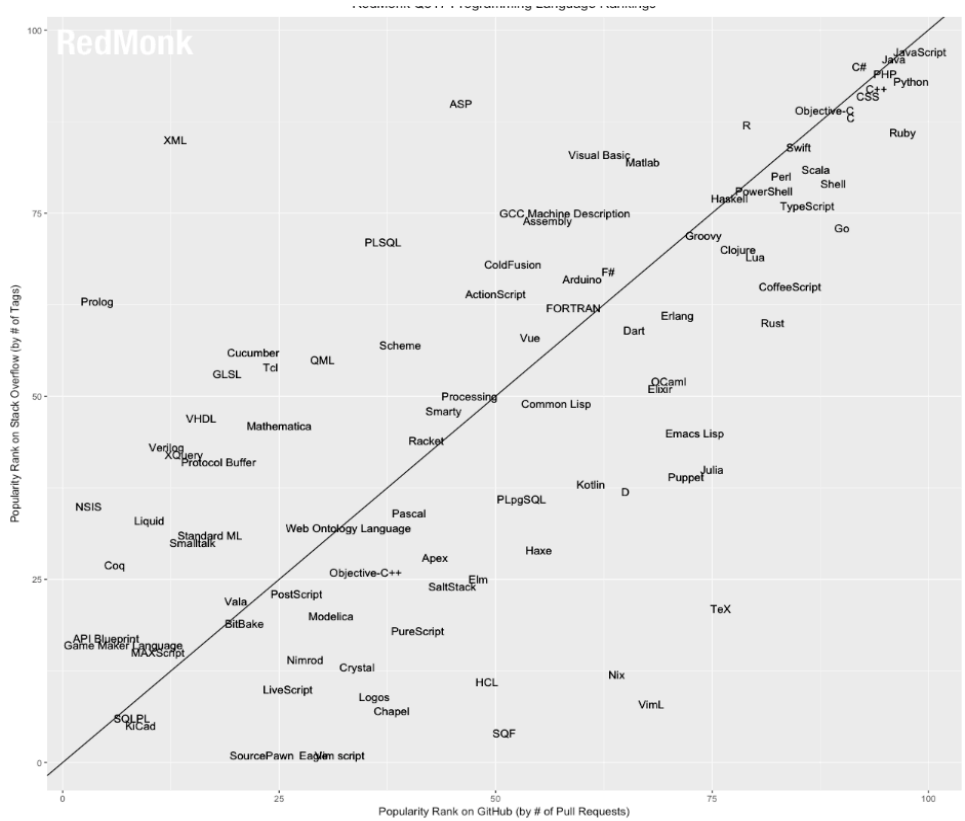
- Most of the APIs shipped in the JDK (java.*, javax.*) have specifications managed under the JCP (Java Community Process)
  - Million and millions of tests to ensure compatibility and upgradeability
  - Vast ecosystem of existing OSS libraries we can use to test compatibility
- To update the specification of such an API, a JCP process is required
  - Rigorous JSR & OpenJDK technical review and assessment process

A Java compatibility review includes, but is not limited to…

- Overall suitability for the platform
  - Recommended use of language features and other libraries
  - Following JDK API conventions
  - Compliance with standing technical policies, including general evolution policy
  - *"Have you considered yesterday, today and tomorrow impacts"*
- Verify proper technical reviewer is present
- Help make connections to other areas
- For some security fixes with behavioral changes we are working hard to find a fix that meets security and compatibility requirements

Java Platform Compatibility Commitment

A big reason for Java being the #1 Development Platform over the past 10 years!



JDK 9 Migration: First Impression

You guys are breaking everything!

JDK 9 Migration: Acknowledgment

No, I don’t want to hear that I need to modify my app!

JDK 9 Migration: Reality

I migrated my app, it was not that bad!

JDK 9 The good news

- The class path has not changed
- Class loading has not changed
- You are not forced to migrate your code to modules
- sun.misc.Unsafe works as before
- Most existing code should work as before
- The IDEs, Maven, … all have support for JDK 9 already

The not so good news

- You will have to upgrade the libraries and tools that you use
- If you use any of the components that are shared between Java SE and Java EE then you may need to adjust your build or deployment
- A small number of supported APIs have been removed
- A number of non-API features and tools have been removed
- You may see some warnings from the libraries that you use

Incompatibilities expected in JDK 9 due to

- Bug fixes
- Changes in supported algorithms
- New security requirements
- Changes in supported platforms
- Use of native code
- Changes in properties
- Deprecations triggering new warnings
- Changes in deployment (browser)
- Change in internal workings
  - E.g. Garbage collection algorithms or
  - Encapsulation of internal APIs
- Change in packaging
  - File names, internal structure of files, registry keys in Windows, installation folders, etc.
- Additional tools or files no longer included
- Removal of deprecated methods, classes, and functionality

#1 Incompatibility expected in JDK 9

A third party library or tool that your code relies on is affected by the changes to JDK 9

- As part of the JDK 9 work we are contacting developers of many common frameworks and common libraries that we have identified as affected
- Expect most vendors with large installed base to provide updates before JDK 9 releases but…
- Until your code adopts the new versions: it is likely affected

JEP 260: Encapsulate most internal APIs

Internal APIs no longer accessible to developers

- The vast majority of internal APIs will become inaccessible by default
- A handful of widely used internal APIs without good alternatives will remain public until a later major release when alternatives are created
- jdeps, tool available with JDK 8, can help identify programs that might be impacted (better to use jdeps from JDK 9 EA as it is more current)
- Multi-Release JAR files (JEP 238) is being introduced to allow many Java-release-specific versions of a class to coexist in a single JAR file
- A command line option will allow reverting this change for JDK 9

JEP 223: New Version String Scheme

JDK 1.9.0_25 turns into JDK 9.1.3 and all the binary filenames change

- The version string for JDK 8 and earlier is convoluted and non-standard
- New version string is closer to Semantic-Versioning and drops some long kept baggage (e.g the "1." in front, the use of "u" for updates)
- Java properties will change; programs that rely on version strings to check for correct Java version might need to be updated
- All filenames for Java binary filenames will also change
  - Build and test systems that assumed old format will need to be updated

JEP 220: Modular Run-Time Images

Bye, bye rt.jar

- With the new module system the internal structure of the JDK changes significantly
- Directory structures and filenames inside the JDK folder are changing
- All programs that inspect the JDK folder and infer anything from the existence, or lack of, any file are likely to be in trouble

Remove most sun.misc.* and sun.reflect.* APIs

Except for a few called out on JEP 260

- In order to keep critical APIs without a replacement accessible by default sun.misc and sun.reflect will not be hidden and a few APIs kept “public”
  - sun.misc.Unsafe
  - sun.misc.{Signal,SignalHandler}
  - sun.misc.Cleaner
  - sun.reflect.Reflection::getCallerClass
  - sun.reflect.ReflectionFactory
- All other APIs in these packages (e.g. sun.misc.Base64) will be removed

JEP 214: Remove GC Combinations Deprecated in JDK 8

- JEP 173 deprecated these GC combinations with JDK 8
- Unsupported and untested since JDK 8
- These flag combinations will now error

- DefNew + CMS : ParNew + SerialOld : -XX:+UseParNewGC

  ParNew + iCMS : -Xincgc

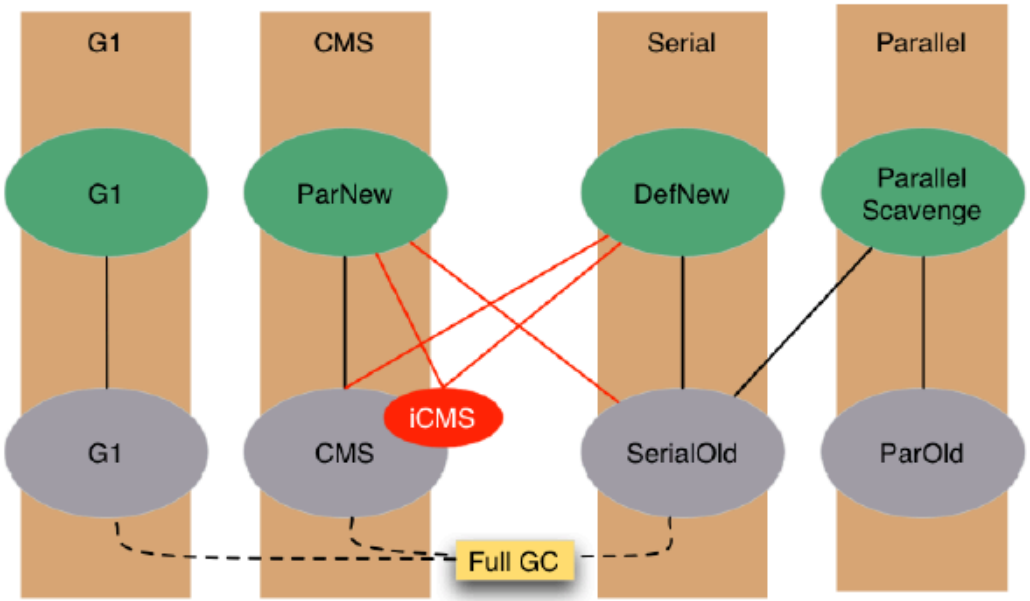
  ParNew + iCMS : -XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC

  DefNew + iCMS : -XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC -XX:-UseParNewGC

  CMS foreground : -XX:+UseCMSCompactAtFullCollection

  CMS foreground : -XX:+CMSFullGCsBeforeCompaction

  CMS foreground : -XX:+UseCMSCollectionPassing



General rule: Look out for unrecognized VM options

- Launching JRE with unrecognized VM options fails
- Using deprecated options in JDK 8 triggers warning messages

```
$ java -XX:MaxPermSize=1G -version
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize; support was removed in 8.0
```

- Previously deprecated options, removed in JDK 9, will fail to start
  - Perm generation was removed in JDK 8 (JEP 122)
  - Expect many programs to run into this problem

Follow up to the work started with JDK 8 in JEP 162: Prepare for Modularization

- Remove addPropertyChangeListener and removePropertyChangeListener
  - From java.util.logging.LogManager
  - From java.util.jar.Pack200/Unpack200
- Remove com.sun.security.auth.callback.DialogCallbackHandle

java.awt.peer and java.awt.dnd.peer packages will be hidden

Non-obvious, but these APIs were for toolkit implementers

- All the functionality these classes provide is available in other, supported, APIs
- jdeps (available since 8 but use the one from 9) will flag these and give you alternatives
- Already announced in OpenJDK, received positive feedback

Thread.stop(Throwable), long time deprecated, will throw exception

http://ccc.us.oracle.com/7059085

- Thread.stop methods are inherently unsafe, cause threads to leave objects in an inconsistent state, and have been deprecated since 1998 (Java SE 1.2)
- Rather than try do do something unsafe it will now throw UnsupportedOperationException
- The no-arg Thread.stop() remains.
  - Although this method is also inherently unsafe, it appears to have significantly more usage than Thread.stop(Throwable)

JEP 271: Unified GC Logging

Re-implemented GC logging with the new JVM logging framework from JEP 158

- Logs are different format
  - Parsers might need to be reworked
- Most common flags are to be deprecated and remapped

  -XX:+PrintGC will be mapped to -Xlog:gc
  -XX:+PrintGCDetails will be mapped to -Xlog:gc*
  -Xloggc:<filename> will be mapped to -Xlog:gc:<filename>

G1 is the new default garbage collector

JEP 248: Make G1 the Default Garbage Collector

- Parallel GC was the default up to JDK 8 on most platforms
  - Still there but no longer the default
  - Can turn back to Parallel GC via command flags
  - Serial GC is, and remains in JDK 9, the default GC on 32 bit Windows

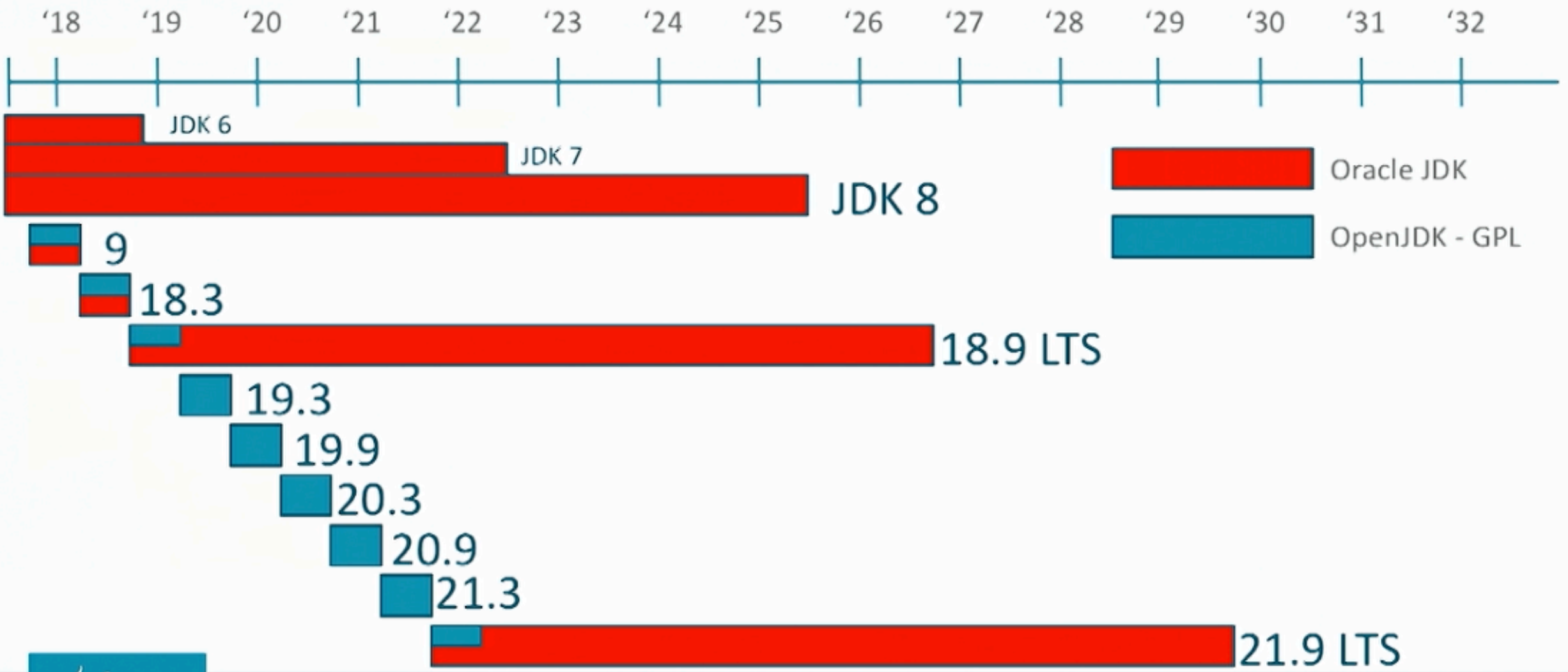
JDK Faster Cadence: 6-month Release Cycle

- Java has developed 9 Major releases over the past 21 years (every ~2.5 Years)
- The Cloud is changing the pace of innovation. Developers want quicker access to new features.
- Under our current model JDK 10 would GA around ***2020***
  - This is forever in Cloud Years!
- From feature bound to time-bound − Every 6 months!
- Uptake of new releases is less of an issue, because less content and cloud operators will do this at scale for you ☺

Simplify the JDK Distribution: OpenJDK vs Oracle JDK

- Large ecosystem with many different constituent profiles
  - Developers -- focused on the bleeding edge, want to see faster innovation
  - Installed base -- maintaining legacy systems, want stability and security
- Need to simplify our distribution
  - Open sourcing commercial features (one code base)
  - Two distribution trains:
    - **OpenJDK** – Focused on delivering innovations at a faster cadence
    - **Oracle JDK** – Stability and critical bug fixes

Oracle JDK & OpenJDK



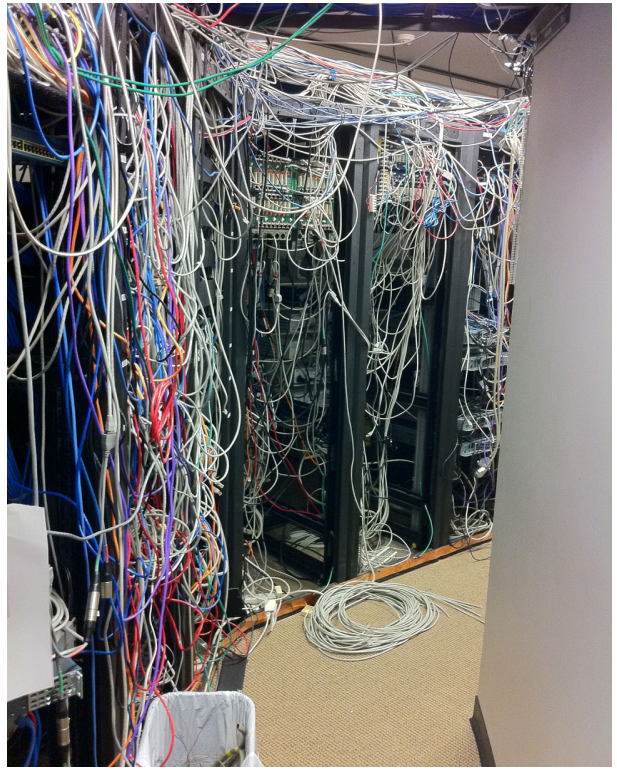
Why Is It Hard to Uptake a New Java Release

- 20% - Removed deprecated APIs
  - Obsolete APIs that have been tagged for deprecation for some time
  - Trade-off of compatibility vs innovation
- 30% Changed in a public API behavior
  - Either the implementation of the spec was wrong, or the spec was wrong and we needed to change the implementation
  - A security fix required to change an unspecified behavior
- **50% Use of internal/non-public APIs**
  - We told you that these APIs may change and they indeed changed

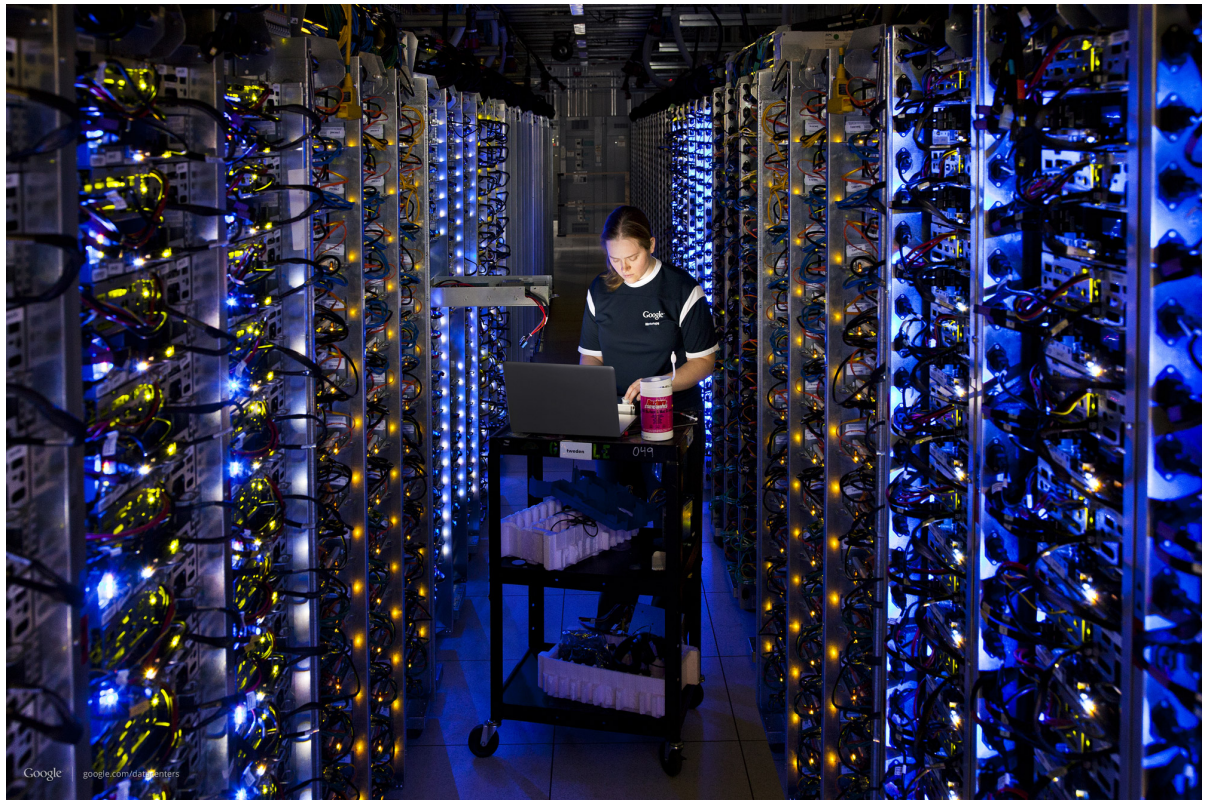
Edsger Dijkstra wrote in 1988,

“We have to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.”

Can We Learn from the Hardware Guys

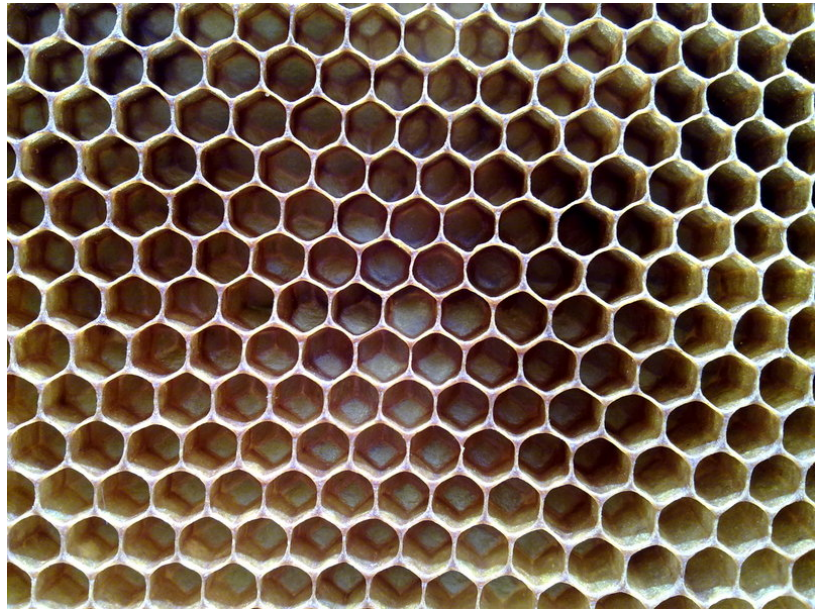


10-15 Years Ago ☹



Data Center Architecture Today ☺

A few Golden Rules



Uniformity

Modularity

Software Modularity & Uniformity For the Cloud

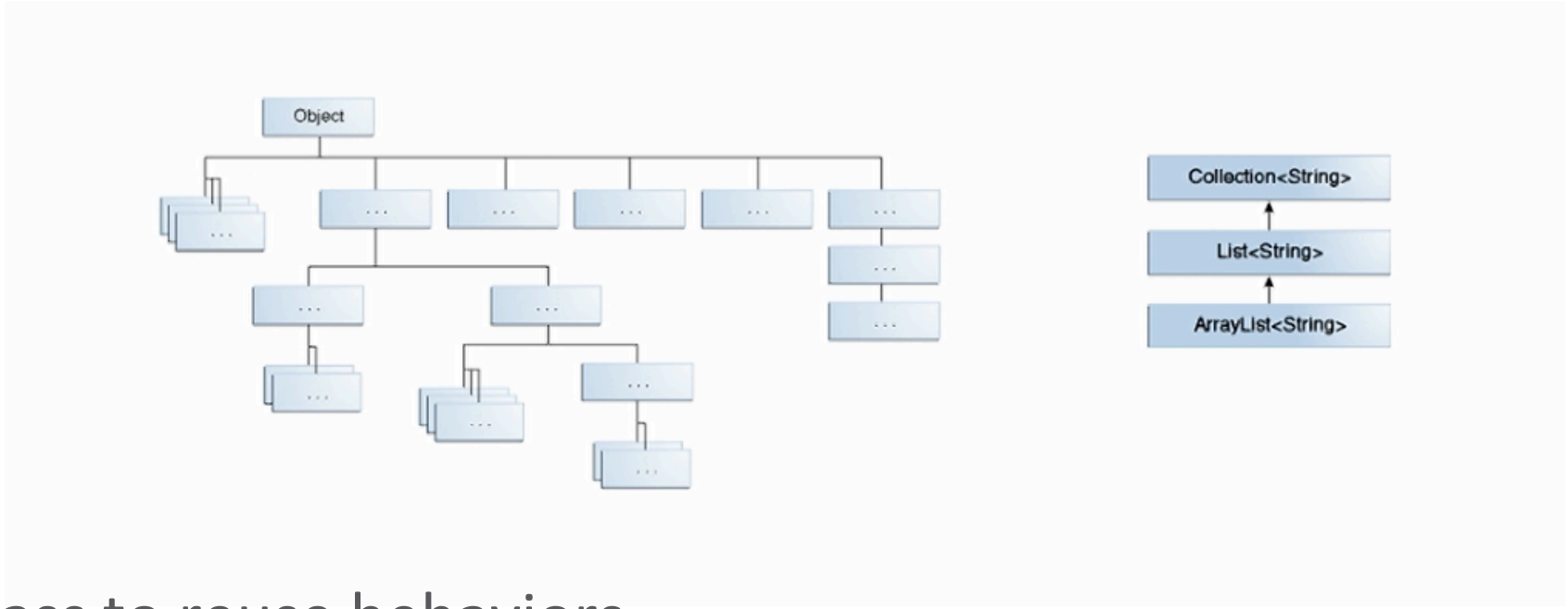
- Docker provides an uniform wrapper around a software package "Build, Ship and Run Any App, Anywhere"
- The container is always the same, regardless of the contents and thus fits on all trucks, cranes, ships, …

- Kubernetes provides an elastic deployment and management

- But, Docker and Kubernetes do not address app modularity inside the container. This is why the Java 9 new module system is so important!

JDK 9 Module

Code Re-Usability and Information Hiding

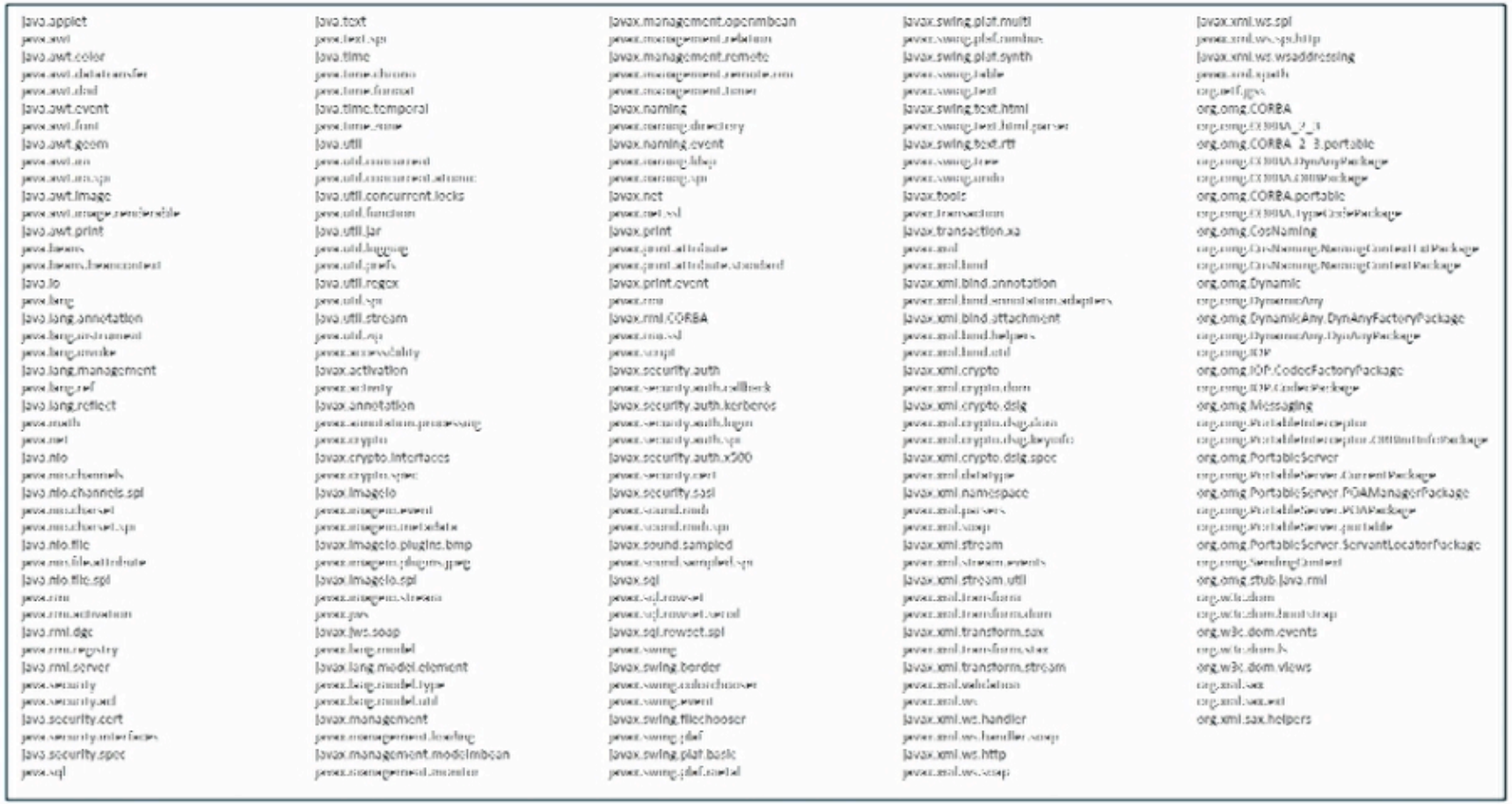


Class to reuse behaviors
Interface to re-use abstractions

JDK 9 Module

Code Re-Usability

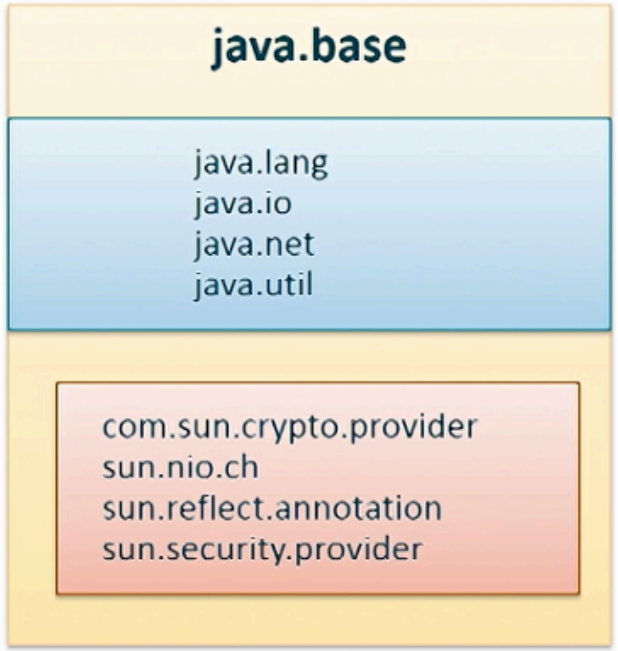
JDK 217 Packages



JDK 9 Module

Java 9 Module

Module is a new fundamental abstraction designed explicitly for encapsulation and reuse




```
// module-info.java
module java.base {
    exports java.lang;
    exports java.io;
    exports java.net;
    exports java.util;
}
```

Java 9 Module – A better Foundation for Compatibility

Access Control Enforced by the JVM

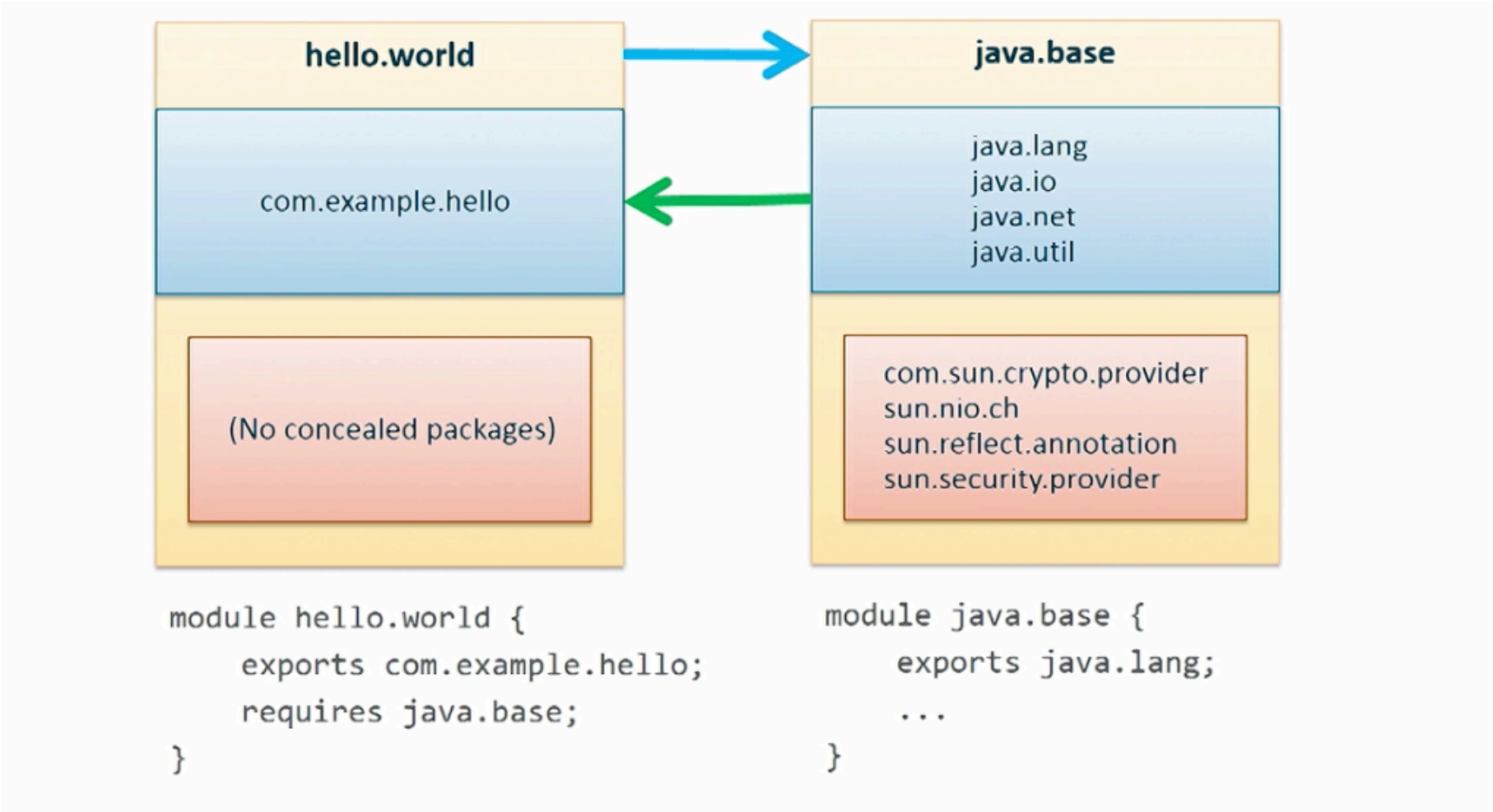
Accessibility (JDK 1 – JDK 8)

- public
- protected
- package
- private

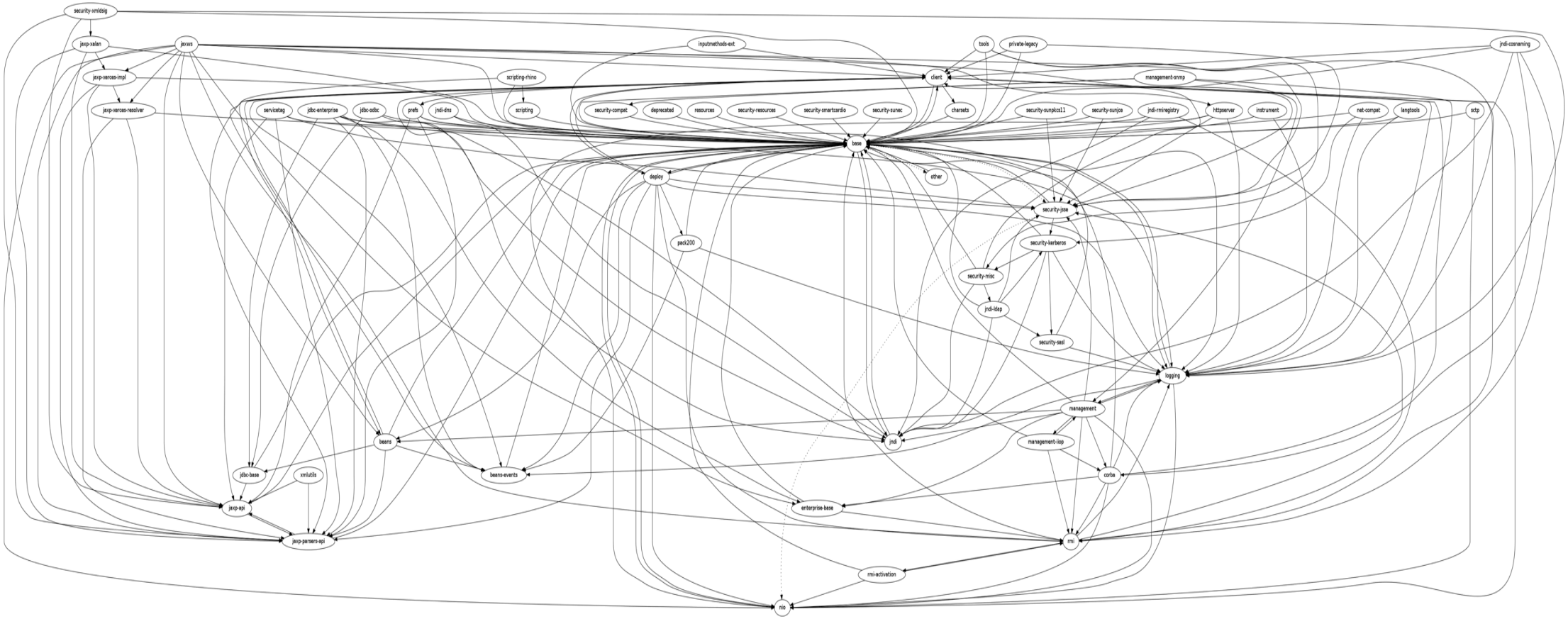
Accessibility (JDK 9)

- *public to everyone*
- *public but only to friend modules*
- *public only within a module*
- protected
- package
- private

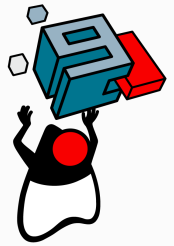
Java 9 Module – A better Foundation for Compatibility

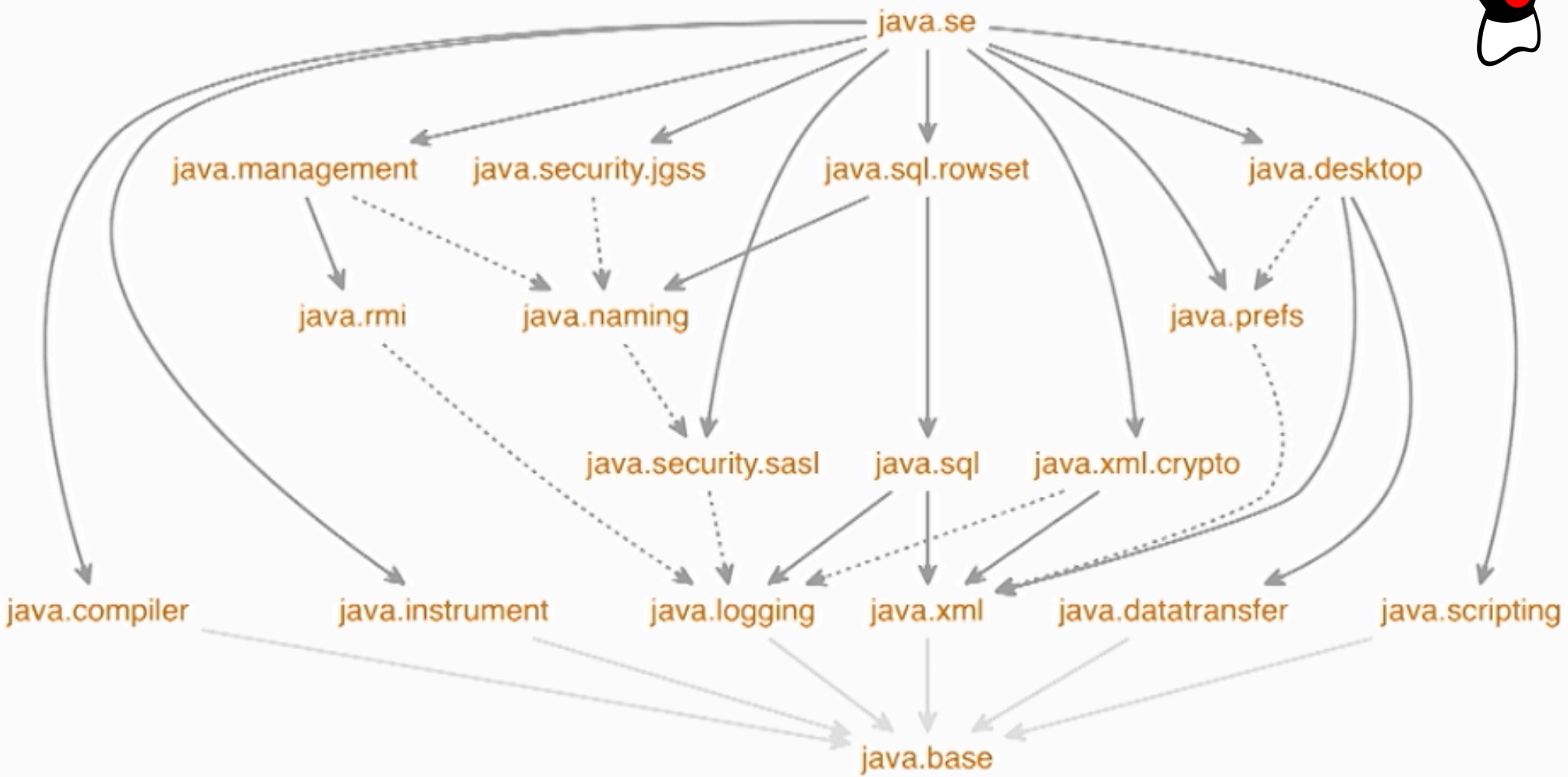


JDK 7 Internal dependencies



The Modular JDK

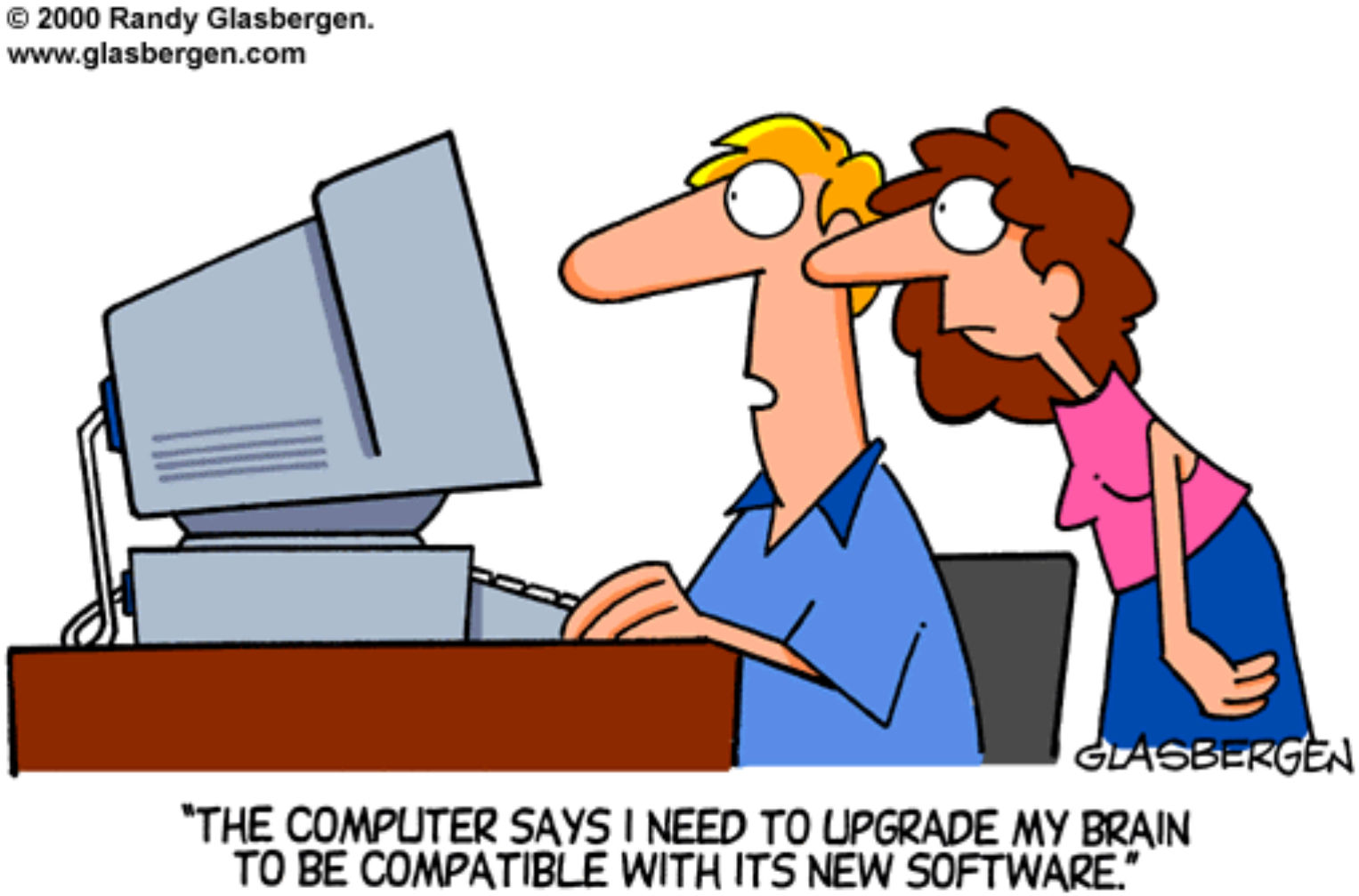




Summary

- The cloud is putting increasing demands on upgradeability (minute/week vs month/year)
  - To reduce complexity and cost (utility computing)
  - To address vulnerability and sustaining upgrades as often as needed
- Upgradeability is significant issues to manage Cloud services at scale
- JDK 9 Module and strong encapsulation at the JVM level is moving the bar beyond what any other developer runtime is providing to improve upgradability and guarantee compatibility
- Uptaking 9 may be some is not that scary!
- Please uptake 9 and assemble your services as module for the sake of our industry ☺

Q&A



JavaOne™

ORACLE®