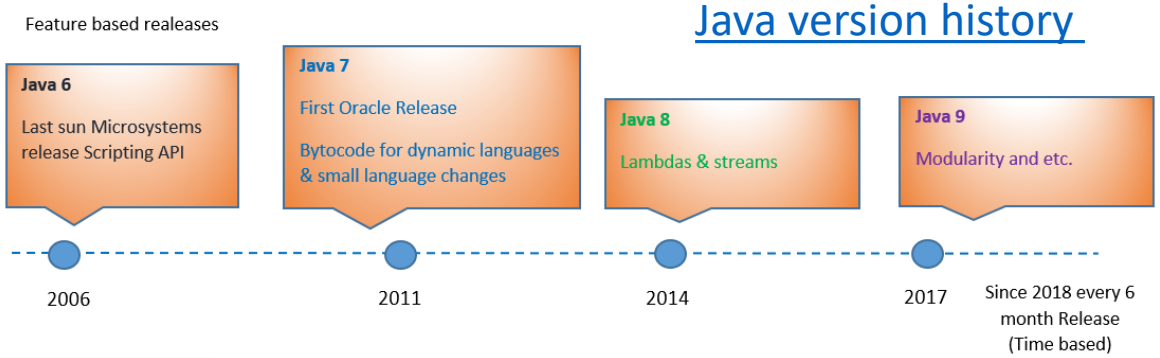
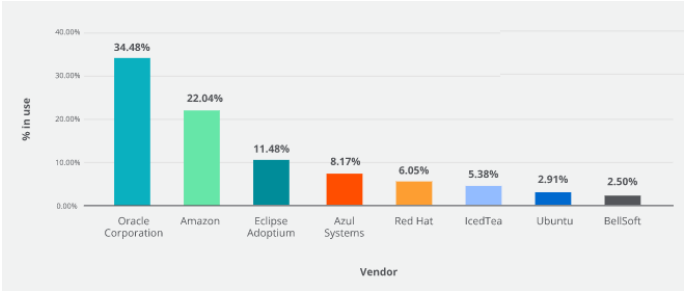
Java version history



Adoption statistics (appr.) in production
in 2020: 84.48% Java **8**, 11.11% Java **11**, in 2022: 46.45% Java **8**, 48% Java **11**
In 2023: Java 17 slightly started to be used in production
Also, note that Java **14** is the most popular non-LTS version
> *java –version*, **>** *java –fullversion*



Java Features

https://github.com/azatsatklichov/Java-Features
http://sahet.net/htm/java.html
https://en.wikipedia.org/wiki/Java_(programming_language)

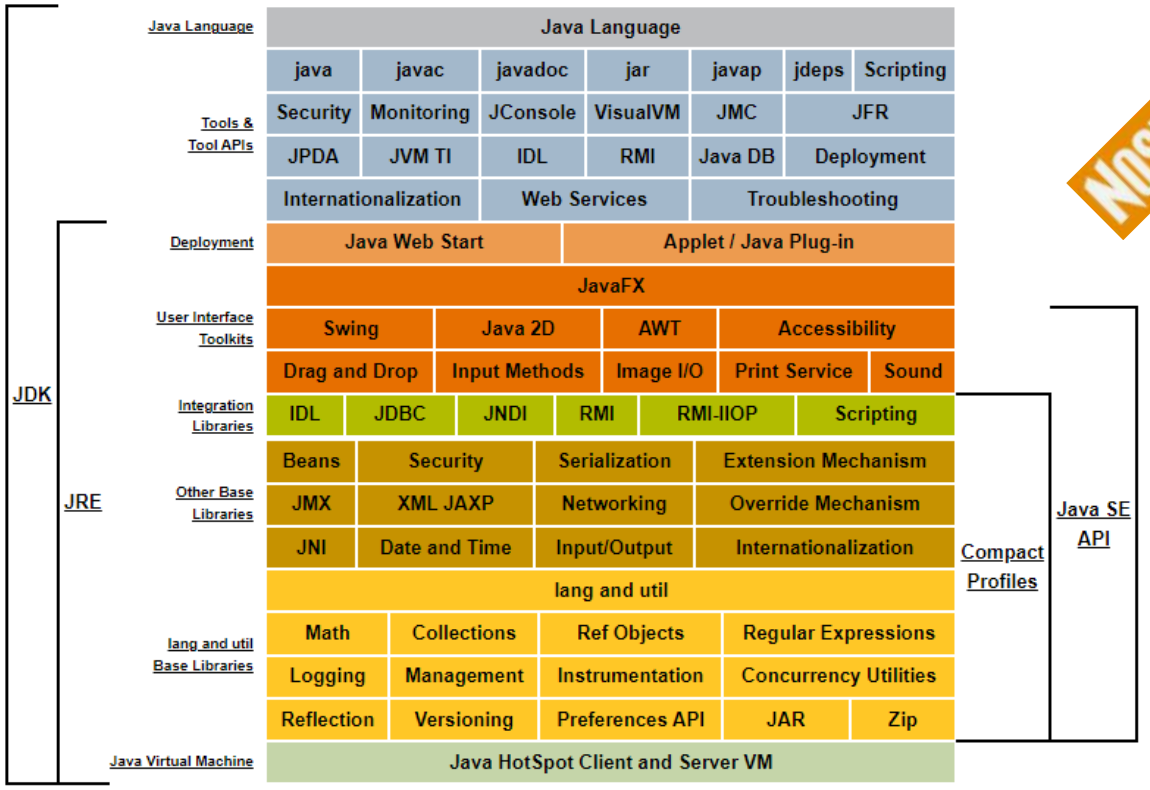
Azat Satklichov
azats@seznam.cz

Sun Microsystems and Oracle

- Java 6 - Last Sun Microsystems release 2006 (Scripting API)
- 27-Jan 2010 Oracle acquired Sun Microsystems - https://en.wikipedia.org/wiki/Sun_Microsystems
- First Oracle release 2011, Java 7 - Bytecode for dynamic languages and some small mall language changes
- 2014 New Era for Java started with Java 8 release – Lambdas, Streams, new Date Time API, and many more

Oak was the initial name for Java given after a big oak tree growing outside James Gosling's window. It went by the name *Green* later, and was eventually named **Java** inspired from Java Coffee, consumed in large quantities by Gosling.

Duke, Oak's smart agent that would later become the Java mascot



JAVA 8 and Beyond

JAVA 10 – Last free version of JDK

Goal: convergence of Oracle & OpenJDK codebases

Oracle JDK 8,9,10
- Binary Code License
- Paid
- Oracle Support Contract

≠

Open JDK 8,9, 10
- GPL v2
- Free

Other option: Amazon Corretto, AtoptOpen JDK (Prebuilt OpenJDK), RedHat Open JDK, ..

Since version 11, **Oracle contributes some commercial features to OpenJDK community**, such as Flight Recorder, Java Mission Control, and Application Class-Data Sharing, as well as the Z Garbage Collector, are now available in OpenJDK. Therefore, **Oracle JDK and OpenJDK builds are essentially identical from Java 11 onward**.

Oracle JDK 11
- Binary Code License
- Paid
- **LTS Support** by Java SE Subscription

=

Open 11
- GPL v2
- Free
- **LTS Support** by Amazon Corretto, AtoptOpen JDK (Prebuilt OpenJDK), RedHat Open JDK ..

Main differences between Oracle JDK 11 (A) and Open JDK 11 (B)

- **(A)** emits a warning when using the *-XX:+UnlockCommercialFeatures* option, with **(B)** builds this option results in an error
- **(A)** offers a configuration to provide usage log data to the "Advanced Management Console" tool
- **(A)** required third party cryptographic providers to be signed by a known certificate, while cryptography framework in **(B)** has an open cryptographic interface, which means there is no restriction as to which providers can be used
- **(A)** will continue to include installers, branding, and JRE packaging, whereas **(B)** builds are currently available as *zip* and *tar.gz* files
- The *javac –release* command behaves differently for the Java 9 and 10 targets due to the presence of some additional modules in (**A)'s** release
- The output of the *java –version* and *java -fullversion* commands will distinguish Oracle's builds from OpenJDK builds

JAVA 8 and Beyond

Since Java SE 10, **every six months there** is a new release. Not all releases will be the Long-Term-Support (LTS) releases, it will happen only in every three years. Java SE 17 is the latest LTS version

Oracle JDK

Contains more tools than the standalone JRE as well as the other components needed for developing Java applications. Oracle strongly recommends using the term JDK to refer to the Java SE (Standard Edition) Development Kit

Open JDK

OpenJDK is a **free and open-source** implementation of the Java SE Platform Edition. Initially, it was based only on the JDK 7. But, since Java 10, the open-source reference implementation of the Java SE platform is the responsibility of the JDK Project. E.g AtoptOpen JDK (Prebuilt OpenJDK), Amazon Corretto, RedHat Open JDK .

Oracle JDK vs. OpenJDK

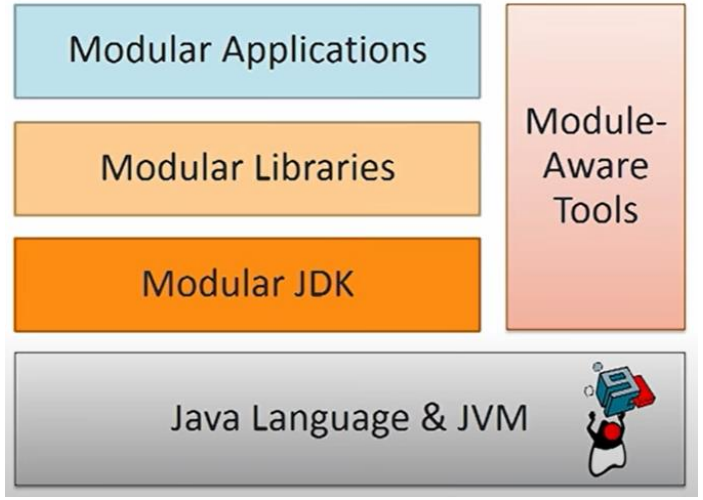
Release Schedule: Oracle will deliver releases every three years (err. ???), while OpenJDK will be released **every six months**. Oracle provides long term support for its releases. On the other hand, OpenJDK supports the changes to a release only until the next version is released.
WoooW LTS 2 years –!!!! - Oracle is proposing that the next LTS release should be **Java 21, Sep. 2023**, which will change the ongoing LTS release cadence from three years to two years, so **in every 2 years after Java 17.**
Licenses: Oracle JDK is under Oracle Binary Code License Agreement, whereas OpenJDK has the GNU GPL version 2 with a linking exception.
Performance: Even though no real technical difference exists in these two, **Oracle JDK is much better regarding responsiveness and JVM performance**. It puts more focus on stability due to the importance it gives to its enterprise customers. OpenJDK, in contrast, will deliver releases more often. As a result, we can encounter problems with instability. Based on community feedback, we know some OpenJDK users have encountered performance issues.

Features: Besides standard features and options, Oracle JDK provides **Flight Recorder, Java Mission Control, and Application Class-Data Sharing features**, [contributed to OpenJDK since Java 11] while OpenJDK has the **Font Renderer feature**. Also, Oracle has more Garbage Collection options and better renderers.

Development and Popularity: Oracle JDK is fully developed by Oracle, whereas the OpenJDK is developed by Oracle, OpenJDK, and the Java Community. However, the top-notch companies like Red Hat, Azul Systems, IBM, Apple Inc., SAP AG also take an active part in its development.
When it comes to the popularity with the top companies that use Java Development Kits in their tools, such as Android Studio or IntelliJ IDEA, the Oracle JDK used to be more preferred, but both of them have switched to the OpenJDK based JetBrains builds.
On the other hand, major Linux distributions (Fedora, Ubuntu, Red Hat Enterprise Linux) provide OpenJDK as the default Java SE implementation.

New Features in JAVA 8 and Beyond, https://github.com/azatsatklichov/Java-Features

- Functional Interfaces, Lambdas Expressions & Method references
- Stream API, New Date-Time API (migrate from java.util.Date -> java.time)
- Process API
- Optional, Collection Factory Methods
- Modular JDK – biggest change ever made to Java (Language, Compiler, VM, Tooling)
- Local var,
- HTTP Client
- Docker Awarene -JVMs are now aware of being run in a Docker container
- New Performance profile and Security features, TLS 1.3, .. updates
- Continious JVM Implrovements: G1 GC, ZGC, Shebandoah, ..
- Tools: FlightRecorder, Java Mission Control, jshell, jDeps, jpackager, jlink, ..
- Language Improvements, Deprecations & Removals
- TLS 1.3, ..
- Records, Switch Expressions, Text Blocks, ..



Source code: https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/in/java8

When Migrating Java to Beyond Java 8

When migration consider 3 points: Modularity (**optional:** still classpath based app can be used **–illegal-access=permit**), Library additions, Security&Performance improvements like Docker awareness, G1 GC, etc. see below

- Build tools: Maven min. version 3.8.0 , <configuration> <release>11</release> </configuration> , Gradle version 5 , or Docker image: maven:3-openjdk-11
- Plugins, .. Eclipse or IntelliJ newer version
- Modularity (internally) – to be prepared to next 20 years
- Deprecated APIs (Base64, SecurityConst, enterprise APIs-CORBA, JAXB,JTA, so use Jakarta EE)
- Removed ones
- Mainly you need to solve, ne to deal with "encapsulated types / jdk internals" and "non-default" module usage
- Use jdeps to finds dependencies, to migrate .. E.g. [> jdeps –jdkinternals MainDemo.class]
- Try run on Java 11 with: --illegal-access=deny (not dependent on jskinternals, e.g. **sun.misc.Unsafe**) – **FUTURE PROOF** code for modularity
- For temporary solution (not full migration) use --add-exports .. --add-modules to solve non-default module , … --permit-illegal-access
- Start using Java 11 features like List.of(), etc.
- LTS to LST is good practice e.g. Java 11 -> Java 17 -> Java 23. But you can use non-LST
- So still aware of non-LST versions (Java 13,14,15,16, 17(next LST), 18, ..), what is new, removed, deprecated, ..
- Plan your FUTURE : --illegal-access=deny
- In case to deep dive in JVM, use visualvm to with installing plugins visualgc and grallvm
- So, simply CLASSPATH to MODULEPATH
- Also note that releasing LTS to LTS is a big gap, a lot features, deprecations/removals may impact, better approach is try upgrading non-LTS versions on extra-dev branch or use Amazon/RedHat/OpenJDK LTS versions for those not supported LTS by Oracle

New Features (JEPs) in JAVA 9

102: Process API Updates
110: HTTP 2 Client
143: Improve Contended Locking
158: Unified JVM Logging
165: **Compiler Control**
193: Variable Handles
197: Segmented Code Cache
199: Smart Java Compilation, Phase Two
200: The Modular JDK
201: Modular Source Code
211: Elide Deprecation Warnings on Import Statements
212: Resolve Lint and Doclint Warnings
213: Milling Project Coin
214: Remove GC Combinations Deprecated in JDK 8
215: Tiered Attribution for javac
216: Process Import Statements Correctly
217: Annotations Pipeline 2.0
219: Datagram Transport Layer Security (DTLS)
220: Modular Run-Time Images
221: Simplified Doclet API
222: jshell: The Java Shell (Read-Eval-Print Loop)
223: New Version-String Scheme
224: HTML5 Javadoc
225: Javadoc Search
226: UTF-8 Property Files
227: Unicode 7.0
228: Add More Diagnostic Commands
229: Create PKCS12 Keystores by Default
231: Remove Launch-Time JRE Version Selection
232: Improve Secure Application Performance
233: Generate Run-Time Compiler Tests Automatically
235: Test Class-File Attributes Generated by javac
236: Parser API for Nashorn
237: Linux/AArch64 Port
238: Multi-Release JAR Files
240: Remove the JVM TI hprof Agent
241: Remove the jhat Tool
243: **Java-Level JVM Compiler Interface**
244: TLS Application-Layer Protocol Negotiation Extension
245: Validate JVM Command-Line Flag Arguments
246: Leverage CPU Instructions for GHASH and RSA
247: Compile for Older Platform Versions
248: Make G1 the Default Garbage Collector
249: OCSP Stapling for TLS
250: Store Interned Strings in CDS Archives
251: Multi-Resolution Images
252: Use CLDR Locale Data by Default
253: Prepare JavaFX UI Controls & CSS APIs for Modularization
254: Compact Strings
255: Merge Selected Xerces 2.11.0 Updates into JAXP
256: BeanInfo Annotations
257: Update JavaFX/Media to Newer Version of GStreamer
258: HarfBuzz Font-Layout Engine
259: Stack-Walking API
260: Encapsulate Most Internal APIs
261: Module System
262: TIFF Image I/O
263: HiDPI Graphics on Windows and Linux
264: Platform Logging API and Service
265: Marlin Graphics Renderer
266: More Concurrency Updates
267: Unicode 8.0
268: XML Catalogs
269: Convenience Factory Methods for Collections
270: Reserved Stack Areas for Critical Sections
271: Unified GC Logging
272: Platform-Specific Desktop Features
273: DRBG-Based SecureRandom Implementations
274: Enhanced Method Handles
275: Modular Java Application Packaging
276: Dynamic Linking of Language-Defined Object Models
277: Enhanced Deprecation
278: Additional Tests for Humongous Objects in G1
279: Improve Test-Failure Troubleshooting
280: Indify String Concatenation
281: HotSpot C++ Unit-Test Framework
282: jlink: The Java Linker
283: Enable GTK 3 on Linux
284: New HotSpot Build System
285: Spin-Wait Hints
287: SHA-3 Hash Algorithms
288: Disable SHA-1 Certificates
289: Deprecate the Applet API
290: Filter Incoming Serialization Data
291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector
292: Implement Selected ECMAScript 6 Features in Nashorn
294: Linux/s390x Port
295: Ahead-of-Time Compilation
297: Unified arm32/arm64 Port
298: Remove Demos and Samples
299: Reorganize Documentation

Biggest changes in Java ever (Language, Compiler, VM, Tooling, …), Language and Library Improvements, New APIs, ..

Source code: https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/in/java9,
And for Java9 see also: ModularJavaProjects

Compare JDK 8 and JDK 9

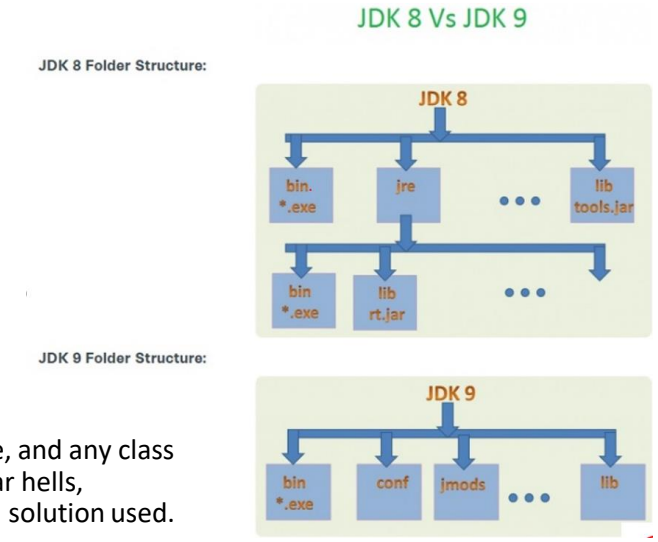
Before Modular JDK

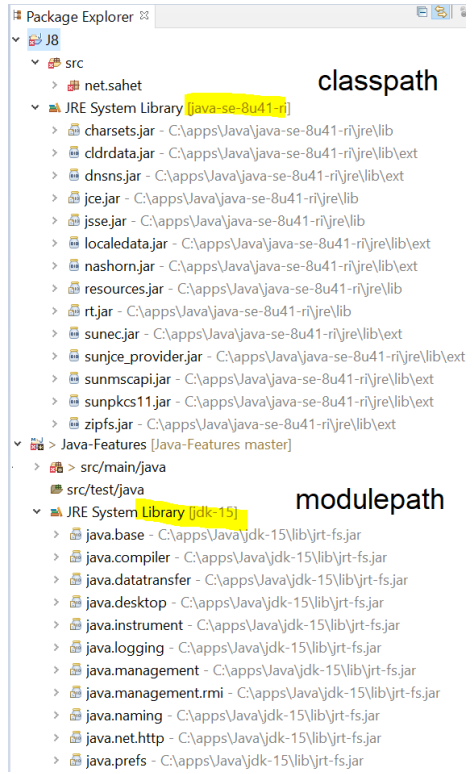
rt.jar

- **one rt.jar (20 yrs. Until 2017)**
- Too big, heavy (entangled classes), technical depth.
- must be backward-compatible, jdk internal classes,
- standard&ee mix)
- CLASSPATH problems [JVM loads all jars into single-threaded CLASSPATH, no name anymore, and any class can access to any class in any jar – open to world – no strong encapsulation] (missing jars, jar hells, ClassNotFound.. …), JAR, EAR, same classpath … but each WAR has separate classpath this a solution used. MODULARPATH solves all above issue.
- Java 8 compact1|2|3 (tries to reduce, but still BIG). Same Docker Java Images but still runtime size is big. Also project OSGi is a way of Modularization, but it is based on current Java. So, Best option is Jigsaw JAVA.

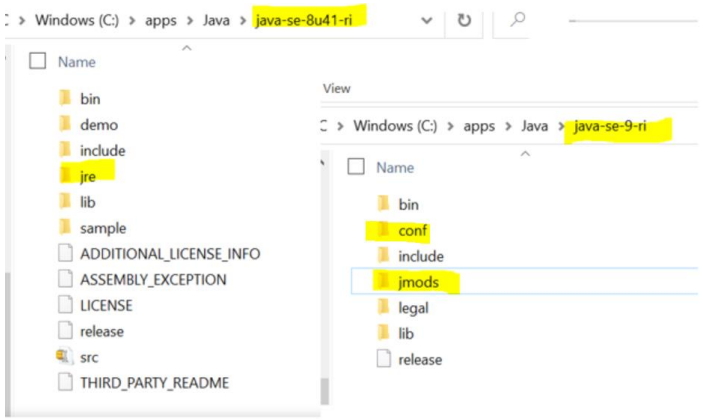
Modular JDK (solving technical depths and brings modularity)

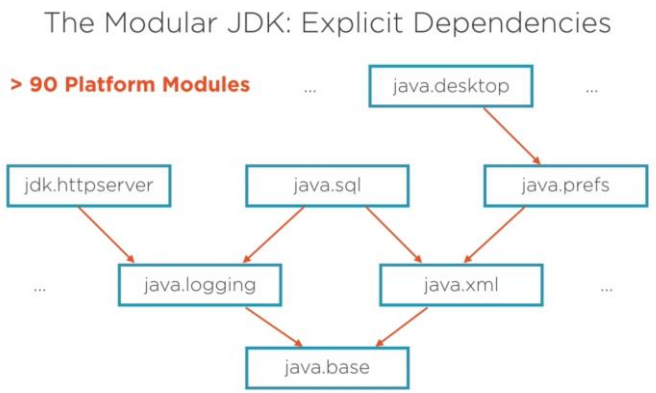
- **JDK modularized, modular apps (than monolithic)**
- Create own modules (modularize apps)
- Using module system is optional (**unnamed module**)





Here JDK 9 does NOT contain JRE. In JDK 9, JRE is separated into a separate distribution folder. JDK 9 software contains a new folder "jmods". It contains a set of Java 9 Modules as shown below.





All modules has implicitly dependency to **java.base** (foundation).
E.g. Remember java Object class.

Modular JAVA (Project Jigsaw [J7])

Jigsaw project is going to introduce completely new concept - **Java Module System**.
JEPs: 200 (modular JDK), 201(modular src-code [rt.jar is too big]), 220(modular runtime images), 260(encapsulate most internal APIs), 261(module system – mod. app), 282(jlink – java linker)

JSR 376 (Jigsaw - Java Platform Module System)

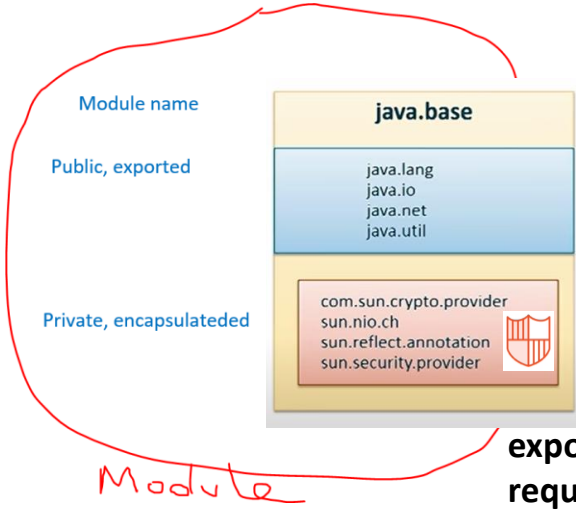
//Java 9 is modular, 11 – JEE modules removed

- **Module system** (module is named, groups related code [data+resources, mod-desc) & self sufficient [supports SRP], designed for re-use)
- **Small** and clear, compact runtime images, reliable config (**no cyclic dep**., explicit dependency info can be used in compile and runtime)
- **Increased security (can't use jdk internals)**, improved **performance**
- 95 < modules (jdk.*, java.* [java spec modules], java.base – default to all)
- Easy of deprecation (java.corba, …), Future **proof** (ship, try incubators)
- **Why jar not** enough (has name but no meaning at Runtime, groups code but no encapsulation – world access [e.g. executable jar put by hacker can destroy your code, security issue], not self contained - lack of explicit dependency in Java [yes by maven OK] but if 'provided' option used and not exist in server lib..)
- 3 cornerstones of modularity: Stronger encapsulation, reliable-config (clear explicit dependencies - via requires clause: no cyclic dependency, no class-not-found issues), well-def. interfaces – public API)
- Module Types: System or Platform, Application (our app), Automatic (for non modularized jars it creates modeule-info.java), Unnamed

➢ **java --list-modules** //system modules

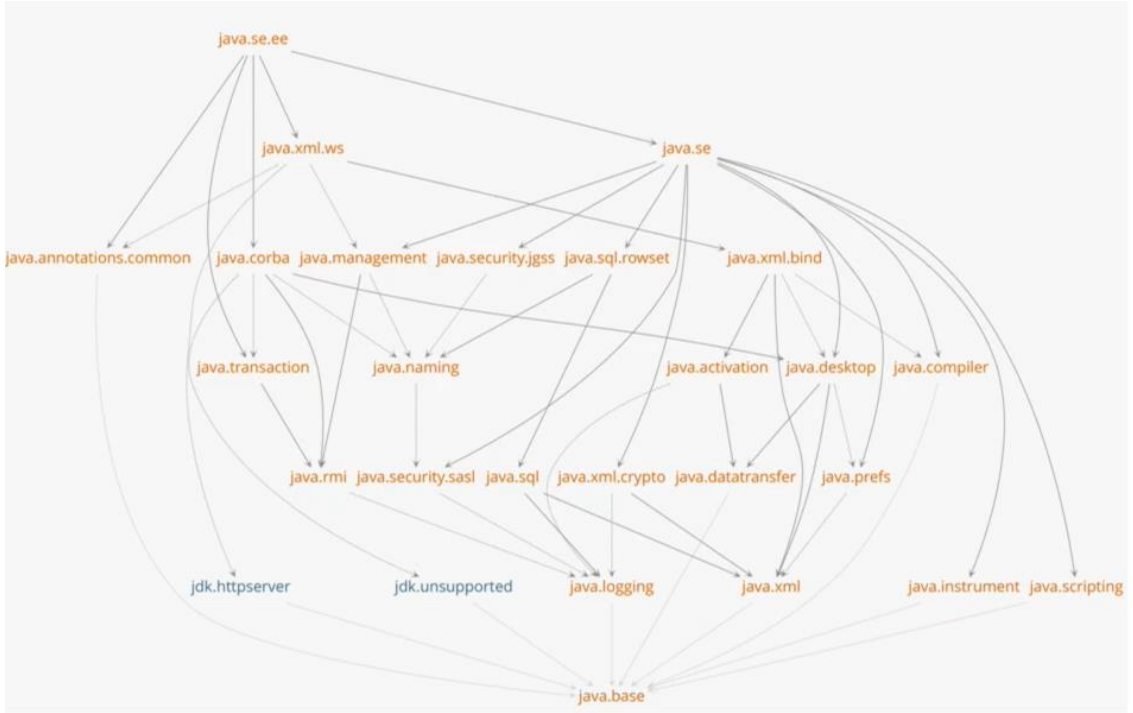
➢ **java --describe-module java.sql**

```
//module-info.java   (module descriptor)
```




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

exports uses "package name"
requires uses "module-name"



Creating Modules

Module contains: one module (cannot have sub-module/s), has unique module name, packages, types, native code (if module has JMOD format), resources and one module descriptor //module-info.java – contains module meta-data

A module descriptor describes a named module and defines methods to obtain each of its components. Modules export packages (API Interface) and require other modules (explicit dependency). Not listed means (hidden) strong encapsulation.

Modules can be distributed one of two ways: as **a JAR file** or as an "**exploded**" compiled project. Format: jmod, jar. **Jmod** – contains native libs. etc… when shipping module with native code.

```
module coremodul {
    exports net.modul.core;
}
```

exports
opens
provides
requires
uses

```
app.client.modul
    module-info.java
    app
        client
            modul
                ClientApp.java
core.modul
    module-info.java
    core
        modul
            Fibo.java
```

exports uses "package name"
requires uses "module-name"

It is common to only have one Java module per project.

Creating Runtime IMAGES

Packing a Java Module as a Standalone Application

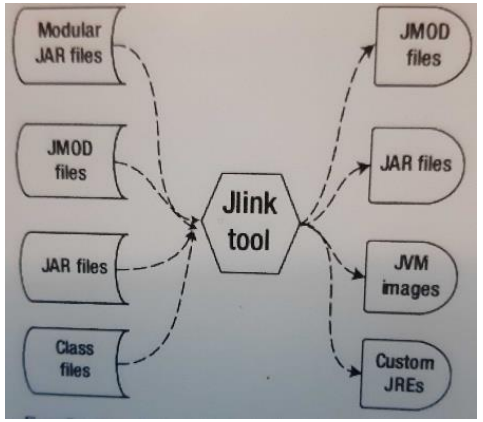
Using **jlink** tool you can package a Java module along with only required modules (recursively) and the Java Runtime Environment into a standalone application [small app. **distribution**]. No Java pre-installation needed to run the application, Java is included in pack. Very small, fast and portable. Compact(custom) runtime image you can deploy to cloud…, or make docker images. (classic way: still some java image+app.)

```
//inspect module definitions
C:\workspace-JavaNew\Java-Features>java --describe-module java.sql
java.sql@17-ea
exports java.sql      //package name
exports javax.sql
requires java.transaction.xa transitive
requires java.base mandated
requires java.logging transitive
requires java.xml transitive          //module name
uses java.sql.Driver
```

```
> jlink --module-path "out;C:\apps\Java\jdk-17\jmods"
--add-modules test.core.modul --output out-standalone
```

Also can be optimized: --strip-debug --compress 0|1|2

```
> cd C:\workspace-JavaNew\J8\out-standalone
echo "Running the image"
bin\java --module test.core.modul/test.core.modul.Fibo

> bin\java --list-modules   ;)
```




```
C:\workspace-JavaNew\J8\out-standalone>bin\java --list-modules
java.base@17
test.core.modul

C:\workspace-JavaNew\J8\out-standalone>
```

```
JDK17 (without app): 293.8 MB
Runtime Image (app+runtime): 40.3 MB
Runtime Image (app+runtime) optimized: 25 MB
```

Migrating Classpath based apps to Module based (modulepath) - Concerns

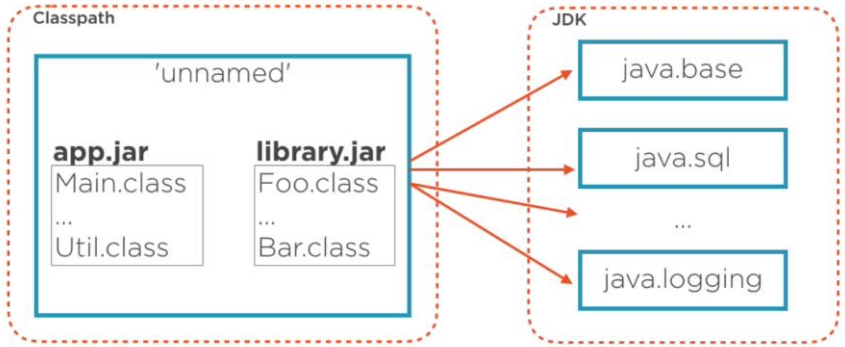
JAVA 8 → 9 (that easy?) > javac -cp $CLASSPATH, java -cp $CLASSPATH

using module system is optional (**unnamed module, automatic m.**), can keep using classpath.

The unnamed module can read all named modules, automatic or non-automatic.

//unless use JDK types 1. encapsulated, 2. non-default java modules. 3. Cyclic, etc.

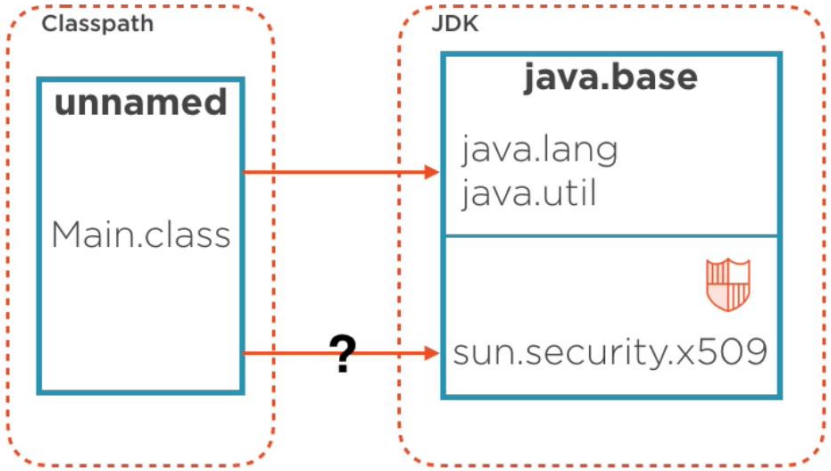
Automatic module - add any library's JAR file to an app's module path, it becomes automatic



1. Using encapsulated types

```
import sun.security.x509.X500Name;

public class MigrationExample { public static void main(String[] args) { X500Name nem = new X500Name("CN=user"); }}
```

```
//apps still allowed to use encapsulated types in JDK for backward comp.

//Runs on Java 8, but strong encapsulation make it fail in Java 9

If you want to focus on future create modular applications

- Code must use public APIs instead of encapsulated internal APIs, use tool jdeps

- Run app with:> -illegal-access=deny [permit /default]  so this is future-proof [not for future, has strong encapsulation, note that =permit  may be removed in future]
```



If you don't want to change the code

> javac **--add-exports** java.base/sun.security.x509=**ALL-UNNAMED** MigrationExample.java

MigrationExample.java:3: warning: X500Name is internal proprietary API and may be removed in a future release

import sun.security.x509.X500Name;

> java **--add-exports** java.base/sun.security.x509=**ALL-UNNAMED** MigrationExample

➢ jdeps –jdkinternals MigrationExample.class

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "1.8.0_41"
OpenJDK Runtime Environment (build 1.8.0_41-b04)
OpenJDK Client VM (build 25.40-b25, mixed mode)

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample.java
MigrationExample.java:3: warning: X500Name is internal proprietary API and may be removed in a future release
import sun.security.x509.X500Name;
```

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample.java
MigrationExample.java:3: error: package sun.security.x509 is not visible
import sun.security.x509.X500Name;
                        ^
  (package sun.security.x509 is declared in module java.base, which does not export it to the unnamed module)
1 error
```

Migrating Classpath based apps to Module based (modulepath) - Concerns

2. Using non-default Java SE modules

```
import javax.xml.bind.DatatypeConverter;
public class MigrationExample2 {
public static void main(String[] args) throws IOException {
DatatypeConverter.parseBase64Binary("EWsaRS43dshfJUir"); }}
```

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "1.8.0_41"
OpenJDK Runtime Environment (build 1.8.0_41-b04)
OpenJDK Client VM (build 25.40-b25, mixed mode)

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample2.java

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>
```

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample2.java
MigrationExample2.java:5: error: package javax.xml.bind does not exist
import javax.xml.bind.DatatypeConverter;
```

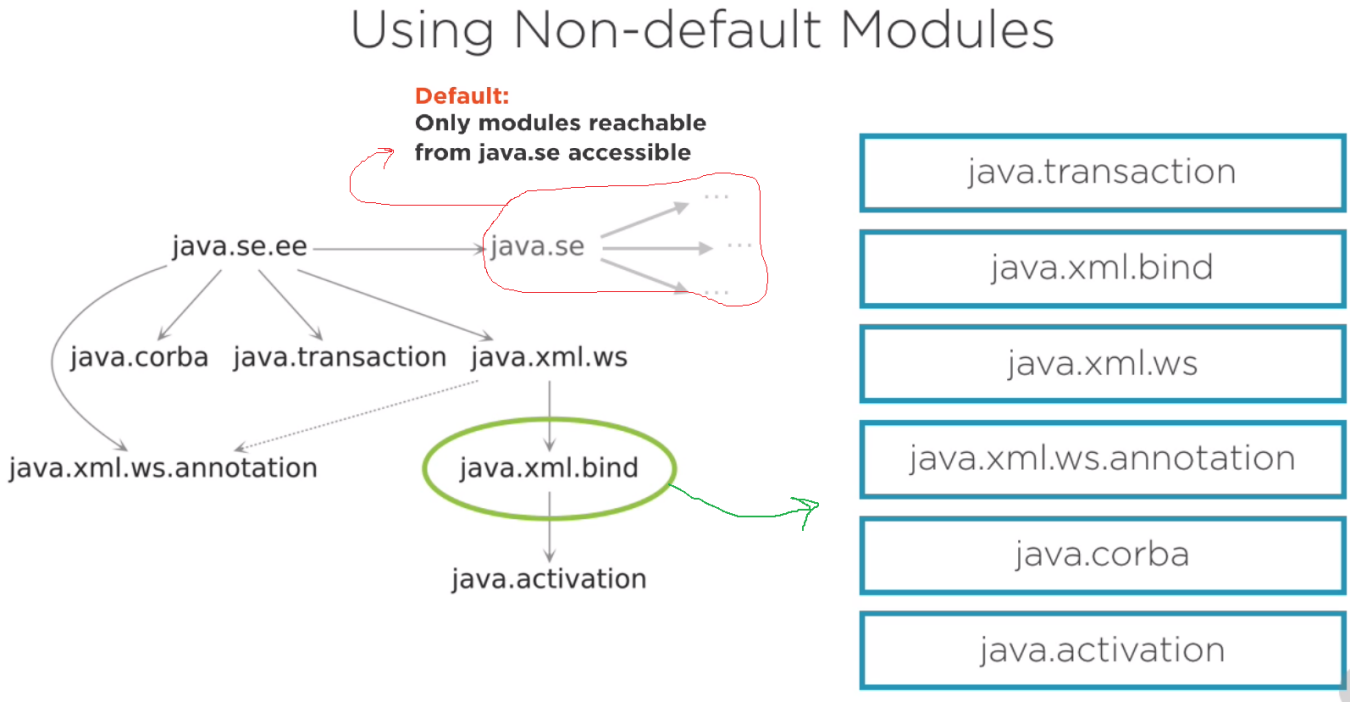
//not reachable, because those libs are app-server specific

implementations, and even removed now, solution:

- javac **--add-modules** java.xml.bind MigrationExample2.java
- java **--add-modules** java.xml.bind MigrationExample2

- java **--add-modules** java.se.ee MigrationExample2
- jdeps MigrationExample2.class

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.0</version>
    <configuration>
        <source>9</source>
        <target>9</target>
        <compilerArgs>
            <arg>--add-modules</arg>
            <arg>java.xml.bind</arg>
        </compilerArgs>
    </configuration>
</plugin>
```



C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>jdeps MigrationExample2.class

Using New Tools in JDK 9

https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

- **jdeprscan** - used to scan usage of deprecated APIs in jar file, classpath, or src-dir.

>jdeprscan.exe --class-path . ProcessApiImprovements_JEP102

class features/in/java9/ProcessApiImprovements_JEP102 uses deprecated method java/awt/List::addItem(Ljava/lang/String;)V

- **jdeps** - analysis your code base (by path to the .class file, dir, jar) list package-wise dependencies and modules. Helpful to migrate to modular-app.

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>jdeps .

- **jlink** - used to select modules and create a smaller runtime image with the selected modules. `jlink – Java Linker`

>jlink --module-path mods/:%JAVA_HOME/jmods --add-modules comp-packt --output img

- **jmod** - is a new format for packaging your modules. This format allows including native-code, config-files, and other data that do not fit into jar-file.

The Java modules are packaged as JMOD files. jmod command-line allows create/list/describe/hash JMOD files.

>jmod ..

jjs - -- removed in Java 15 (….)

- **Diagnostic tools:** Jcmd, jinfo, jps, jmap, jstack, jstat, https://visualvm.github.io

- **jshell** - JEP 222: Jshell (REPL – **R**ead, **E**val, **P**rint, **L**oop) similar to NodeJS, List, Groovy, Scala has. Easily, quickly test new ideas, new features, review API, small POC. Just on CMD. No need to write java class, compile,run. Support code completion (TAB-TAB), analyze, build-in doc. Auto exception-handling on checked ones, but you can do it explicitly.

jshell> "aaaz c".matches("[a]+z\sc")

```
/help   /vars  /imports /methods /types /save myCode.jsh,  /open myCode.jsh , /open  MyDemo.java >jshell –-class-path …
/set feedback verbose
>jshell --class-path commons-lang3-3.12.0.jar       jshell> import org.apache.commons.lang3.StringUtils
jshell>  StringUtils.leftPad("Broadcom", 33)
Jshell can be used as an API for (code completion, source code analysis, ..) IDEs can use it,. Or Jshell js = Jshell.create(); js.eval..
```

New Features in JAVA 9

Language/library Improvements

Collection Factory Methods

- OLD ways: E.g. OldWayOfCreationOrConversion (Java), Apache, Guava, ..
- List<Integer> integers = Arrays.*asList(2, 4, 1, 3, 8, 4, 7, 5, 9, 6, 8);*
- List<String> ooo = new ArrayList<>() {{ add("try workaround1 ");add("try workaround2 "); }};
- Collections.*singletonList(new String("ds"));* Collections.*emptyList();* Collections.*unmodifiableList, ….* Stream.*of(Java8)*

Factory Methods Java 9 (immutable collection) – **list, set, map**, …

- List.of(), List.of(E e1), List.of(E e1, E e2), .. Upto 10 elements.
- Also **static <E> List<E> of(E... elements) { //**intermediate arr. allocation
- List.of(1).getClass -> java.util.ImmutableCollections$List1, 2, 3, N

- Map.of(K key1, V val1) upto 10 pair, Map.ofEntries(Map.Entry<K, V> … entries) //unbounded
- Iteration order for Maps and Sets here not guaranteed, use List for guaranteed order

New Features in JAVA 9

Language/library Improvements

Stream API Improvements

```
//prevent NullPointerExceptions
Static: Stream.ofNullable() (before check and use Stream.empty())
Stream.iterate()[hasNext predicate prm]

takeWhile()[while true],
dropWhile()[drops until true]
```

New Collectors

Advanced collector: Collectors
.groupingBy, `//like SQL query case`
Two new: .filtering, .flatMapping

```
public static <..> Collector<..> filtering(
    Predicate<..> predicate, Collector<..> downstream)

public static <..> Collector<..> flatMapping
        (Function<..> mapper, Collector<..> downstream)
```

```
List<String> ll = getList();
Stream<String> stream =  ll == null ? Stream.empty(): ll.stream();

int count = Stream.ofNullable(ll).count();
System.out.println(count); //0
//or Apache CollectionUtils
Apache - collections4.CollectionUtils.emptyIfNull(list).stream()
//Java8 Optional.ofNullable
stream  = Optional.ofNullable(ll).orElse(List.of()).stream();

Stream.of("a", "b", "c", " ", "d", "", "z").takeWhile(s ->
!s.isEmpty()).forEach(System.out::print); //abc d

Stream.of("a", "b", "c", " ", "d", "", "z").dropWhile(s ->
!s.isEmpty()).forEach(System.out::print);//z

Stream.of("a", "b", "c", " ", "d", "", "z").takeWhile(s ->
!s.isBlank()).forEach(System.out::print);//abc
Stream.of("a", "b", "c", " ", "d", "", "  ", "z").dropWhile(s ->
!s.isBlank()).forEach(System.out::print);// d  z

IntStream.iterate(0, x -> x < 100, i -> i + 4).forEach(System.out::print);
//04812162024

Map<Integer, List<Integer>> ints = Stream.of(11,22,33,63) .collect(groupingBy(i -> i % 2,toList()));
// {0=[22], 1=[11, 33, 63]}

Map<Double, Set<String>> byPriceMap = winners.collect(
groupingBy(Fifa::getPrice, flatMapping(b -> b.getWinner().stream(), toSet())
)
);
```

New Features in JAVA 9

Language/library Improvements

`of(), empty()`

```
//transform
Optional<String> s = Optional.of("Ola");
s.map(String::toUpperCase); // "OLA"
Optional<Integer> i = Optional.empty();
i.map(n -> n + 1); // still empty, NPE?
```

Optional Improvements

`ifPresentOrElse(), or(), stream()`

```
//transform
Optional<String> s = Optional.of("Ola");
s.map(String::toUpperCase); // "OLA"
Optional<Integer> i = Optional.empty();
i.map(n -> n + 1); // still empty, NPE?
```

Optional.stream()

Interoperability between Optional and Stream

```
List<Optional<String>> list =
Arrays.asList(Optional.empty(), Optional.of("A"),
Optional.empty(), Optional.of("B"));

List<String> filteredListJava9 = list.stream()
.flatMap(Optional::stream).collect(Collectors.toList(
)); //[A, B]

optional.ifPresentOrElse(x ->
System.out.println("Value: " + x), () ->
System.out.println("Not Present.")); //[A, B]

Optional<String> optional1 = Optional.of("Rimini");
Supplier<Optional<String>> supplierString = () ->
Optional.of("Not Present");
optional1 = optional1.or(supplierString);
optional1.ifPresent(x -> System.out.println("Value: "
+ x));//Value: Rimini
```

New Features in JAVA 9

Small Language Changes

JEP 213 – Milling Project Coin

- Private methods in Java Interface

- Underscore as identifier illegal: String _ = "underscore"; //as of release 9, '_' is a keyword
  Possible future use: list.forEach(_ -> doSomething())

- Improved try-with-resources

//Direct instantiation, previous Java version.
//Before Java 9, you have to declare it

```
public void normalTryWithResources() throws IOException {
    try (FileInputStream fis = new FileInputStream("~/tmp/test")) {
       fis.read();
    }
}
```

//in practice you already have f.i.s. And want to use it in try(), in Java 9 OK
//condition is fis must be **effectively FINAL**

```
public void doWithFile(FileInputStream fis) throws IOException {
    // fis = null; // Re-assignment makes fis not 'effectively final'
    try (FileInputStream fis2 = fis) {
        fis2.read();
    }
    // Only if fis is 'effectively final', can this form be used
    try (fis) {
        fis.read();
    }
}
```

- Better generic type inference for anonymous Java compiler improved to support this

```
// JAVA 9
Handler<Integer> intHandlerz = new Handler<>(1) {
    @Override
    public void handle() {
        System.out.println(content);
    }
};
```

- Localization: Unicode 8.0, java.locale.providers=COMPAT,CLDR

- Java Time: **Duration#dvidedBy(),truncatedTo(), Clockn#systemUTC(), Locale#datesUntil()**

New Features in JAVA 9

Other Improvements

- NEW Java9 Javadoc HTML5 way (html output, search, modules)

Old-api: https://docs.oracle.com/javase/8/docs/api/
>javadoc -d C:/JAVA_doc_new -html5 _IntroJava9.java
New-api: https://docs.oracle.com/javase/9/docs/api/overview-summary.html

- Localization

Unicode 8.0: 10000+ new characters, Properties-files: ISO-8859-1 to UTF-8
Common Locale Data Repository (JDK now reach locale data from standard cldr formats)
If you want old way, use: java.locale.providers=COMPAT, CLDR, …

- java.time since Java 8

Added methods: Duration#dividedBy(), truncatedTo(), Clock#systemUTC() //most precise, nano
LocalDate#datesUntil()

- `@Deprecated(forRemoval = true, since = "9")`

e.g java.lang.Object#finalize, avax.net.ssl.HandshakeCompletedEvent#getPeerCertificateChain

New Features in JAVA 9

Language and Library Improvements New APIs

- **`JEP 238 Multirelease:`** Allows developers to build jars with diff. version of class files for diff. Java ver.
  - `javac --release 7 JRelease7.java`
  - `javac --release 9 JRelease9.java`

- **JEP 102**: ProcessHandle – Improves Process Handling

- **Process** represents native process created by Java, **ProcessHandle** represents any process on OS
- **Process#toHandle**, ProcessHandle.of(123), **ProcessHandle.Info** //list processes, and more info like task mgr. in OS
- Long.parseLong(ManagementFactory.getRuntimeMXBean().getName().split("@")[0]); //process id before Java 8
- ProcessHandle.current().pid(); //process id by Java 9

- **JEP 110**: HTTP/2 Client (incubator in Java 9, default in Java 11) – replacement for HTTPURLConnection, supports http/2
- **JEP 266** (more concurrency update): java.util.concurrent.Flow, Flow API: Publisher, Subscription, Subscriber

Reactive Streams – stream data support for backpressure. Interfaces to impl. pub-sub: RxJava, AKKA Streams, Spring 5

- **StackWalker** (new API to support navigating the stack trace, more helpful than just printing stack trace)

- Before: StackTraceElement[] stackTrace = new Throwable().getStackTrace(); //or stackTrace = Thread.getStackTrace();
- Low performance, No guarantee all stack elements are returned, No partial handling possible
- Java 9: StackWalker, StackFrame. StackWalker walker = StackWalker.getInstance(); walker.forEach(System.out::println);

New Features in JAVA 9

Desktop Enhancements

- Java Browser Plugin Removed
- Deprecated: **java.applet.Applet** (use **Java Web Start** as a fallback solution), or JS handles all solution now.
- Extended HiDPI support, Graphics Improvements – GTK+3 support on Linux
- Marlin Renderer (OpenJDK 9) better than Oracle Ductus, OpenJDK Pisces
- New API for taskbar interaction
- Platform Specific Desktop Features:
- java.awt.Desktop new methods
- java.awt.desktop new package with interfaces: *Callback types for events and handlers*
- java.awt.Taskbar manipulate taskbar icon, show progress, manage context menu.. isSupported()

- Multi Resolution Images: Represent same image at different resolutions

- JavaFX Modularized: jfxrt.jar (splitted into modules) – Control, Skin, Behavior. Java FX is most modern GUI
- JavaFX new APIs: java.fx.scene.control.skin, javafx.css

- Platform Specific Desktop Features:

New Features in JAVA 9

- GC Deprectaions
- GC combinations deprecated in Java 8 removed
- CMS (Concurrent Mark Sweep) garbage collector is depreciated. Use: -XX:+UseConcMarkSweepGC
- G1 – Garbage First garbage collector (introduced in Java 6, now became default in Java 9, replaces CMS)

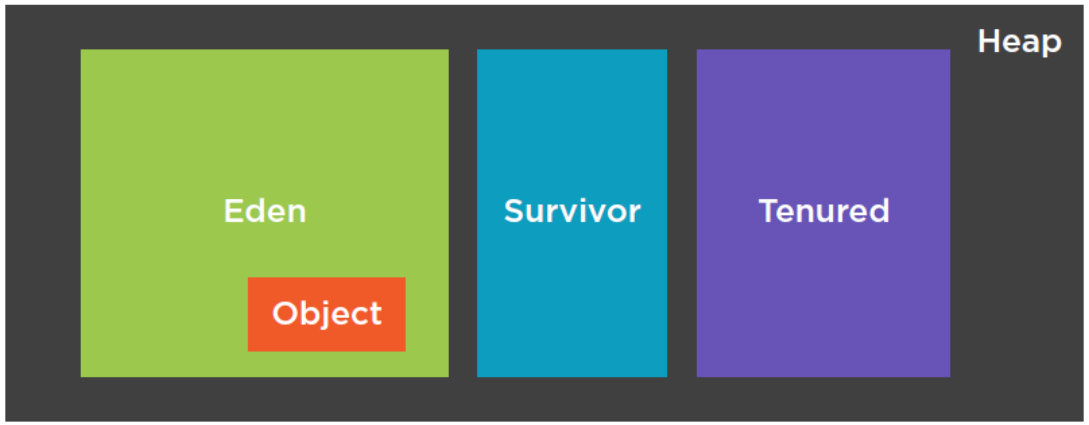
- Generational Garbage Collection (issue is tuning, long pause "STOP THE WORLD") [Heap regions: Eden, Survivor, Tenured]
- G1 Garbage Collector [Heap divided multiple: Eden, Survivor, Tenured regions, as matrix]
- G1 is incremental, Parallel marking, Designed for large heaps, a bit more CPU intensive, Low pauses, tunable pause goal
- Automatically tunes: Heap region size, Parallel threads, pause time interval

- Short and **faster** memory management with: -**XX:MaxGCPauseMillis=200** //tune pause goals
- Trade some throughput for lower latency: -XX:+**G1EnableStringDeduplication** //performance optimization

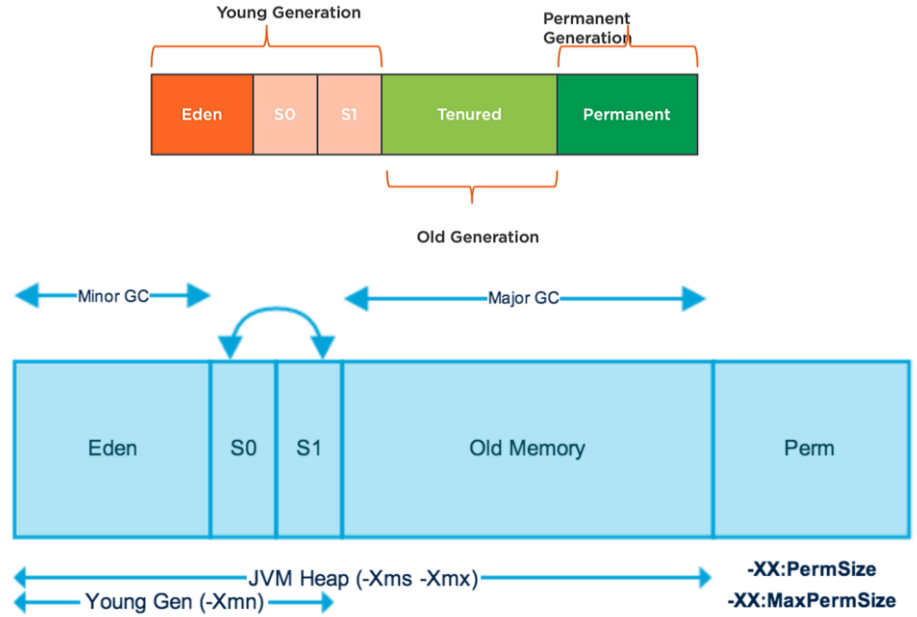
JVM Performance Improvements

- G1 (Garbage First, or **compacting-collector**) Garbage Collector – incremental GC. Paper 2004, experimental in Java 6, official in Java 7, and default GC since Java 9. Designed for apps running on multi-cores with large memory space. Goal is: **Througput/Latency balance.** Tuneable pause goals. A bit more CPU intensive. Regions (#2000, size 2MB for 4GB heap)
- Evacuation (moved/copied between regions). Minor & Major collection,..
- >java -XX:+UseG1GC -jar App.java

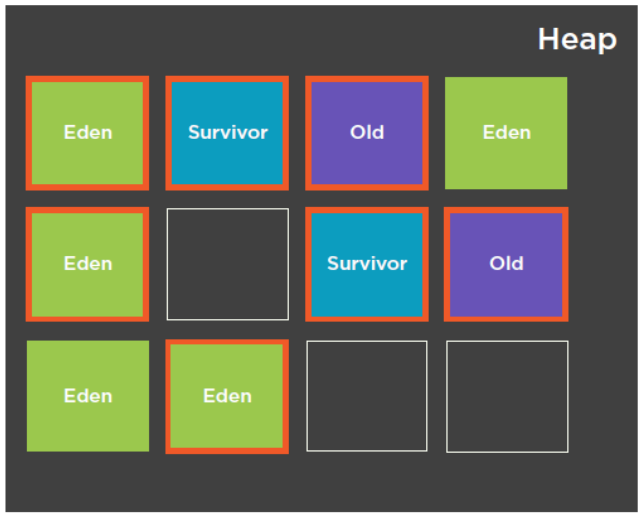
Generational Garbage Collection



Long 'stop-the-world' GC pauses
Difficult to tune



G1 Garbage Collector



Incremental GC

Parallel marking

Designed for large heaps

Low pause, tuneable pause goal

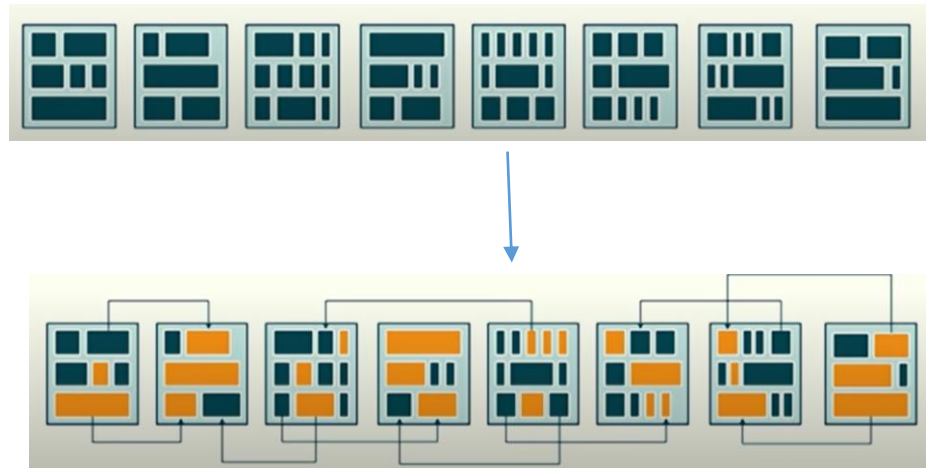
Slightly more CPU intensive

Automatic tuning:

Heap region size

Parallel threads

Pause time interval



New Features in JAVA 9

String Performance Improvements

- Compact Strings: Lower memory usage, effective immediately without code changes.
  Pre: char[], utf-16 (each char 2 bytes), but for ascii text single byte is enough. **Java9**: byte[], and byte coderFlag
- String concatenation changes: Change concatenation translation strategy, Groundwork for future improvements

- String s = "a" + "b" + "c"; before via StringBuilder#append … recompile ..
- Java 9: invokedynamic bytecode, late binding (at runtime), stable byte code, open to future enhancement.

Danger of Serialization

- ObjectOutputStream, ObjectInputStream -> Vulnerabilities, embedded ..
- New interface – filter data before Serialization: ObjectInputFilter[UNDECIDED,ALLOWED,REJECTED]
- Per stream: ObjectInputStream::setObjectInputFilter, for all streams: ObjectInputFilter.Config.setSerialFilter
- `Or use 'jdk.serialFilter' system property. maxbytes,maxarray, maxdepth = n`
- `So if your app uses serialization, this is the fastest way to secure it, ..`

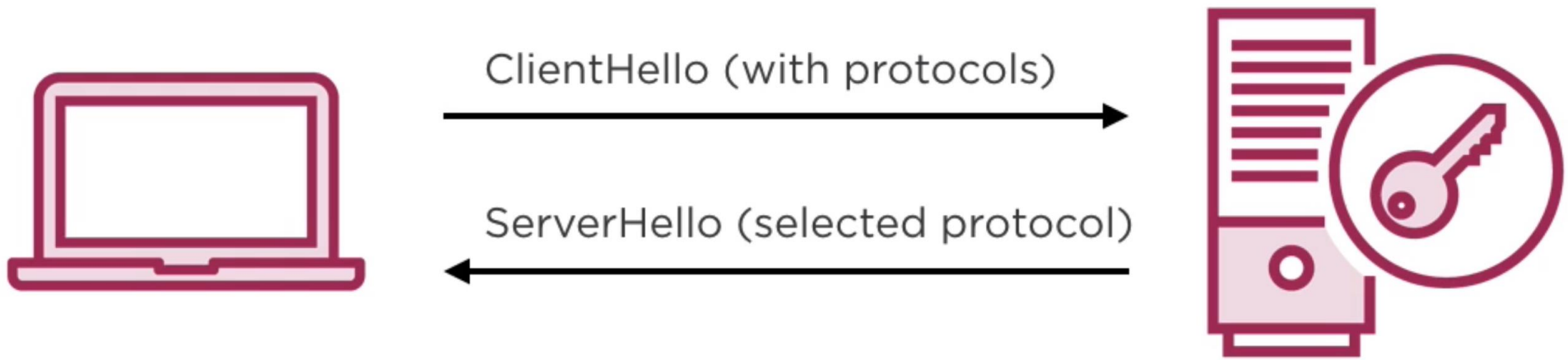
Security Improvements

- TLS improvements (https): ALPN (Application Layer Protocol Negotiation) Select ALPN during Handshake
- Required for http/2 support, httpclient uses it
- Java 9 supports DTLS (Datagram Transport Layer Security) 1.0/1.2 aligned with TLS 1.1 and 1.2 via SSLEngine and DatagramSocket
- OCSP (Online Certificate state Protocol) **Stapling**: check for x.509 certificate revocation when establishing TLS connections
- SHA-1 Certificate is disabled now, SHA-3 support is added instead. Java 9 rejects certificates signed by SHA-1

TLS Improvements: ALPN

Application Layer Protocol Negotiation

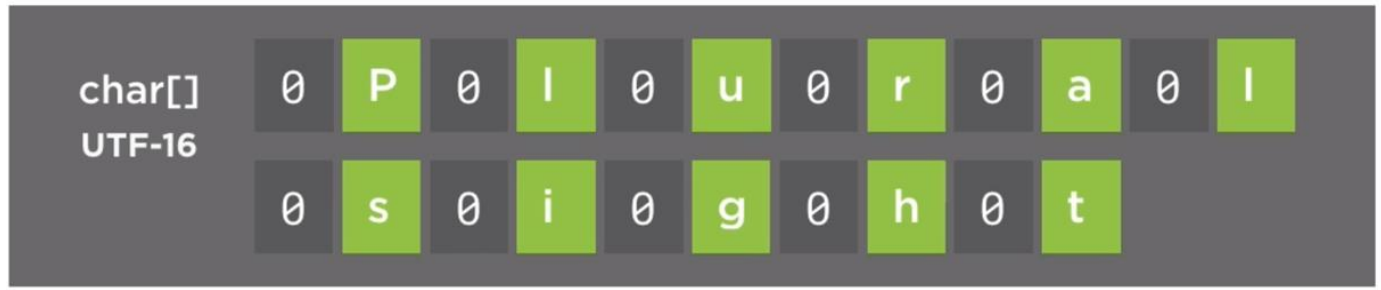
Select application layer protocol during TLS handshake



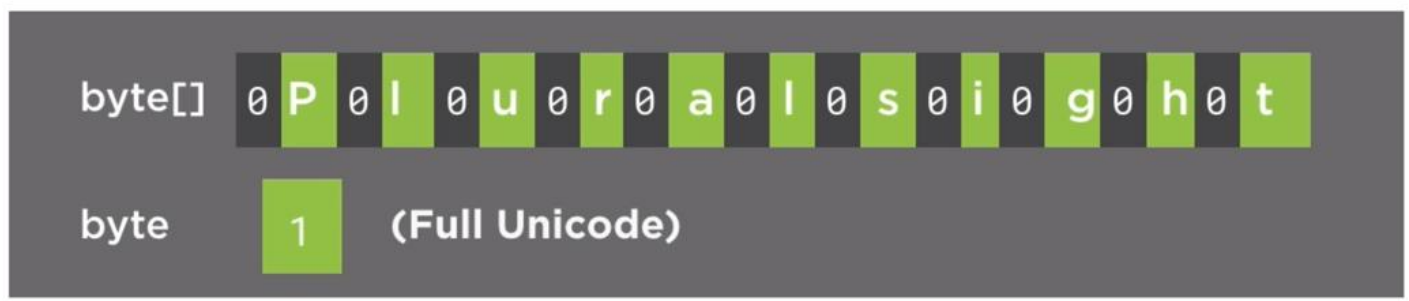
Required for **HTTP/2** support

Compact Strings

Pre-Java 9



Java 9



New Features in JAVA 10

Features

286: Local-Variable Type Inference
296: Consolidate the JDK Forest into a Single Repository
304: Garbage-Collector Interface
307: Parallel Full GC for G1
310: Application Class-Data Sharing
312: Thread-Local Handshakes
313: Remove the Native-Header Generation Tool (javah)
314: Additional Unicode Language-Tag Extensions
316: Heap Allocation on Alternative Memory Devices
317: Experimental Java-Based JIT Compiler
319: Root Certificates
322: Time-Based Release Versioning

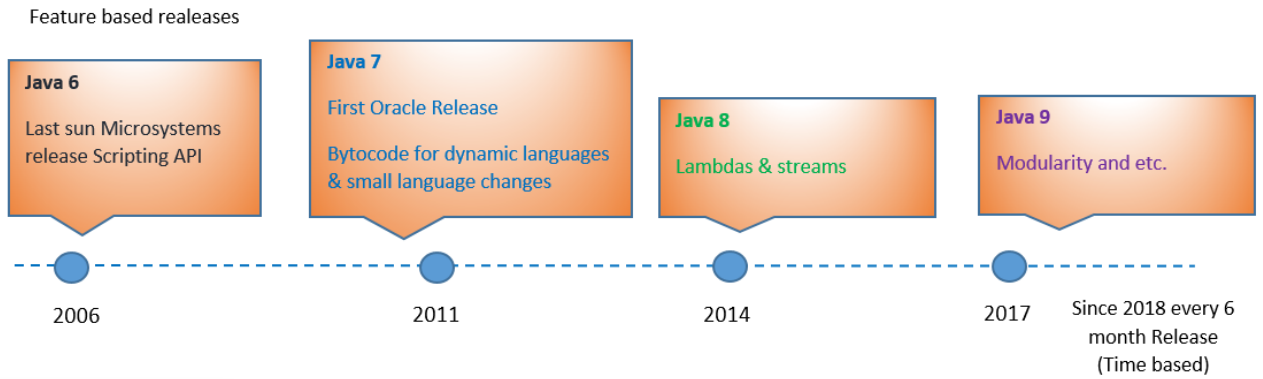
Source code: https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/in/java10

New Features in JAVA 10

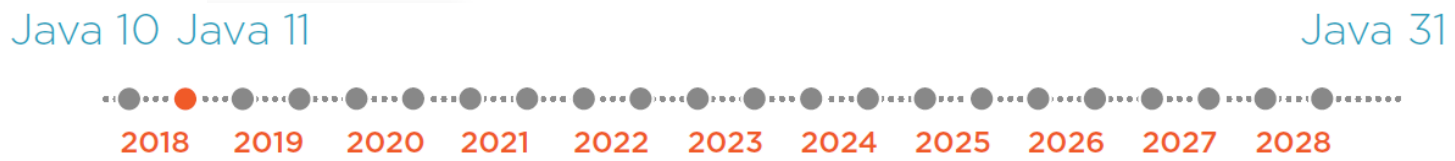
I further propose to aim explicitly for a regular two-year release cycle going forward. Mark R. 2012, [Project Jigsaw: Late for the train]

"I propose that after Java 9 we adopt a strict, **time-based model** with a new feature release every six months." Mark R. 2017
From **"Big feature-driven"** release to **"Time-based"** model

JAVA 10 – Last free version of JDK



New JAVA Release Schedule



Since Java SE 10, in every 6 months there is a new release. **Keeps small and incremental**. Not all releases will be the **Long-Term-Support** (LTS) releases, LTS will happen only **in every 3 years**. Current LTS version of Java is 11, next one is Java 17. Oracle is proposing that the **next LTS** release should be **Java 21**, Sep. **2023**, which will change the ongoing LTS release cadence from 3 years to 2 years, so **in every 2 years after Java 17.**
Convergence Oracle JDK and Open JDK (commercial features will be moved to Open JDK e.g. [ACDS, Flight Recorder(telemetry), Mission Control, Z-GC] Oracle JDK will be commercially supported build of Open JDK code base). Deprecations and removals in JDK 10. **1) Binary build by Oracle only** for WinOS|Linux|MacOS|(no more ARM), 64-bit only, GPL 2 license. Azul(+32 bit+ARM devices), IBM, AdoptOpenJDK has their own BUILDs.
2) Security and Certificates: With Java 10, Oracle has open-sourced the root certificates. Oracle's Root CA program to be ported to OpenJDK. (e.g. SymDump issue)

What about the support? Via LTS (Java 17, Java 23 (now Java 21), …)

Impact of Frequent Release cycle?

- Tool vendors, libraries, frameworks adoption - Releases are small and incremental (can shift) - Non LST releases are standard release

- Deprecations, removals between LTS release, may impact (once migration, 3 (2) years of tech. depth) if you only adopt LTS release. Better differentiate "prod" and "test/dev" environment. Even easier to track with Java Docker images. And, also via –release Java 9 multi-release flag we can keep going, …

New Features in JAVA 10

Root certificate - **cacerts:**

With Java 10, Oracle has open-sourced the root certificates. Oracle's Root CA program to be ported to OpenJDK.

```
See all public certificates :)  > cd C:\apps\Java\jdk-14\lib\security  > keytool -list -keystore cacerts    > * changeit
```

Deprecations and removals

- **javah** command-line-tool is removed. Use alternative: **javac -h <dir>**
- **Policytool** is removed. Even simple text editors can be used to create/edit policy files in case need
- Removed command-line tools/options: **-X:prof** -> If you want profiling info, use jmap tool, or 3rd party profilers
- Deprecated APIs (keep in mind during migration):
  java.security.acl -> java.security.
  In java.security package types {Certificate,Identity,IdentityScope,Signer} are deprecated
  javax.security.auth.Policy -> java.security.Policy

Local-variable (var is not a keyword, also using var is compiled time inferred by **RHS rule**, no runtime overhead)

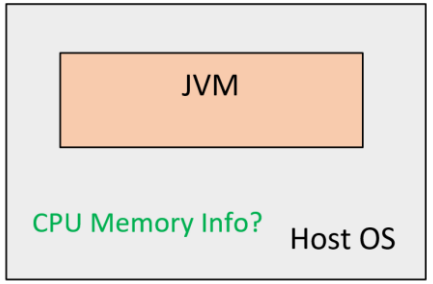
- Local-variable Type Inference [already Java7 <> LHS, or lambdas] like C#, Scala, Kotlin. Also Lombok has “var”, “val” as like
- Focus on variable names (readability), less code, vertical alignments (cursor ;). Driving the meaning of variable.
- var not makes Java dynamic. Just during compilation helps us, inferred by compiler only.
- There are rules, how to use when to use “var”. Not handy to use all the time (not make someone to **GUESS TYPE**).
- Not use var with lambda combination. Not use var with diamond operator. Not use on chained-methods.
- Non denotable types: Null, Anonymous class instances, Intersection types (Comparable & Serializable)
- var empty = null; //NO. var obj = new Object(){} //Object$1 extends Object
- var ls = List.of(1, “2”, 4.67); //List<? extends Comparable & Serializable <..>> //NO

New Features in JAVA 10

JVM Performance Improvements

- G1GC: look Java 9. In Java 9 G1GC is serial full GC (still may cause stop-the-world worst case scenario)
- Java 10, G1GC became parallel full GC: **-XX:ParallelGCThreads**

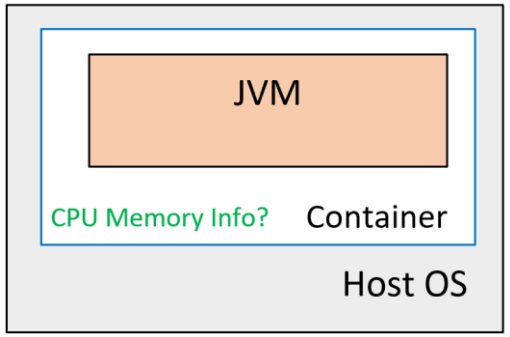
Before Container Awareness



Improved Docker Container Awareness (Linux & Docker only):

- Java now detects container
- Query to container instead of OS (before even run in container queries to OS)

```
-XX:ActiveProcessorCount=<n>
-XX:InitialRAMPercentage
-XX:MaxRAMPercentage
-XX:MinRAMPercentage
```



Optional class improvement:

orElseThrow - java.util.Optional, java.util.OptionalDouble, java.util.OptionalInt and java.util.OptionalLong each got a new method orElseThrow()

```
OptionalInt opInt = OptionalInt.of(12345);
System.out.println("int Value via " + " orElseThrow() = " +
opInt.orElseThrow());//NoSuchElementException


//existing one
IntStream.of(1, 2, 3).average(//).orElseThrow(() -> new
IllegalArgumentException("Array is empty"));
```

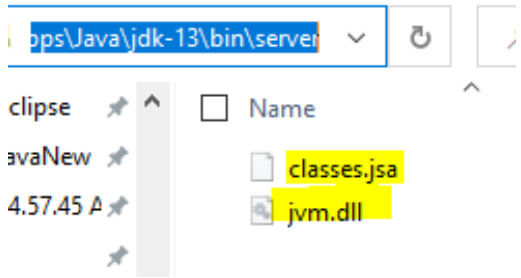
Unmodifiable Collectors:

Unmodifiable collections – **copyOf**() on List, Set, Map and Collectors.**toUnmodifiableList**()

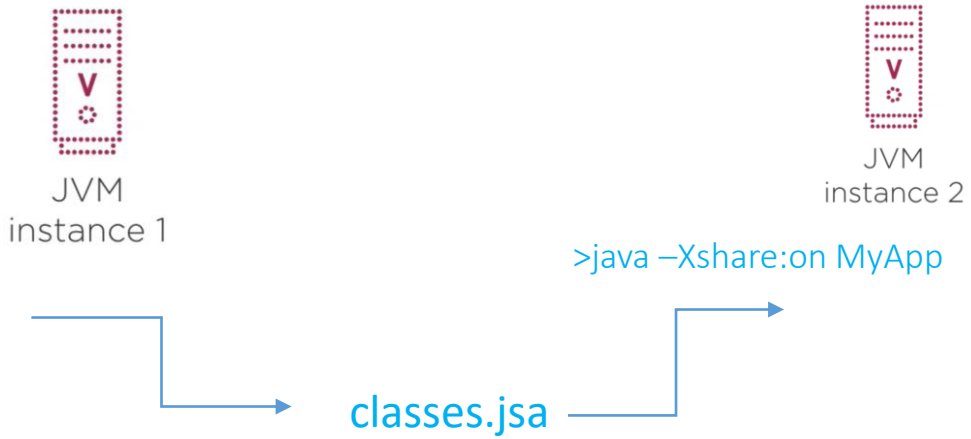
```
//MAP.put(entry.getKey(), ImmutableSet.copyOf(entry.getValue()));
MAP.put(entry.getKey(), Set.copyOf(entry.getValue()));

List<Integer> evenList = ints.stream()
.filter(i -> i % 2 == 0).collect(Collectors.toUnmodifiableList());
evenList.add(4);// UnsupportedOperationException
```

Application Class Data Sharing – ACDS (later in Java 13 improved)

- Goal is to Improve VM start-up time (via avoiding repeated expensive class loading, parsing, verifications, …)
- Sharing (via persisting) the class meta-data info during start-up. Before for system classes only, now app. classes also
- Improve re-boot time on repeated runs(executions) of same JVM, or among multiple JVMs.
- System Class Data Sharing: classes.jsa (saves upto 10-30% startup time)
- Application Class Data Sharing: myApp.jsa, need a extracted class list first: -XX:DumpLoadedClassList=myApp.lst

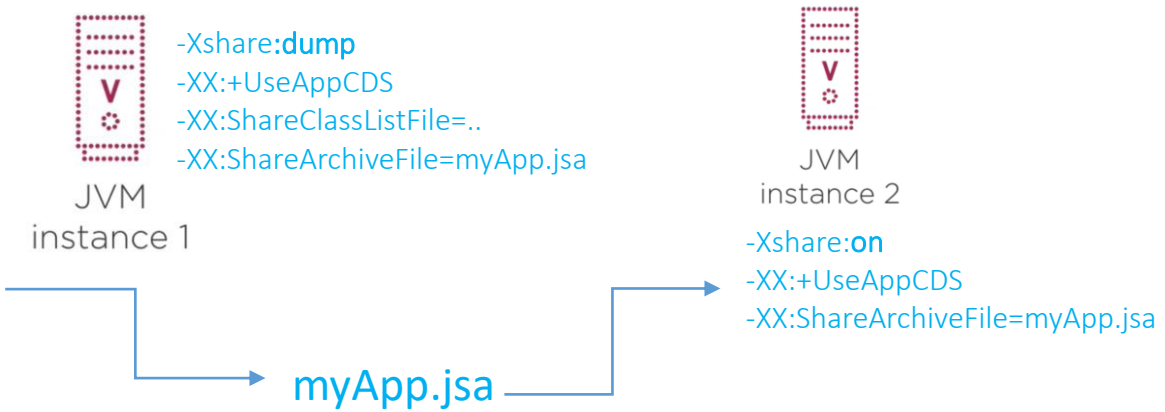


System Class Data Sharing



Improves 10-30% start-up time

Application Class Data Sharing



Startup faster, can get lower memory footprint

New Features in JAVA 11

First LTS Release (Only by Oracle) after Java 8(2019), next Java 17

See slide: JAVA 8 and Beyond

Since version 11, **Oracle contributes (starting from Java 9) some commercial features to OpenJDK community**. Oracle JDK "commercial features" such as Flight Recorder, Java Mission Control, and Application Class-Data Sharing, as well as the Z Garbage Collector, and also Oracle has open-sourced the root certificates in Java 10, are now available in OpenJDK. Therefore, **Oracle JDK and OpenJDK builds are essentially identical from Java 11 onward**. There is no real technical difference between the two since the build process for the Oracle JDK is based on that of OpenJDK. OpenJDK, in contrast, will deliver releases more often.

What about the support? Via LTS (only Oracle and AdoptOpenJDK offers LTS)

Oracle is proposing that the **next LTS** release should be **Java 21**, Sep. **2023**, which will change the ongoing LTS release cadence from 3 years to 2 years, so **in every 2 years after Java 17. See ROADMAP**

Other vendors (Open JDK, RedHat, Azul) has support just for 6 mo. inc. Java 11, 17, … So no difference to keep Java 10 or 11 on Open **JDK** from support point of view.

Goal: **convergence** of Oracle & OpenJDK codebases – DONE, removed: -XX:+UnlockCommercialFeatures

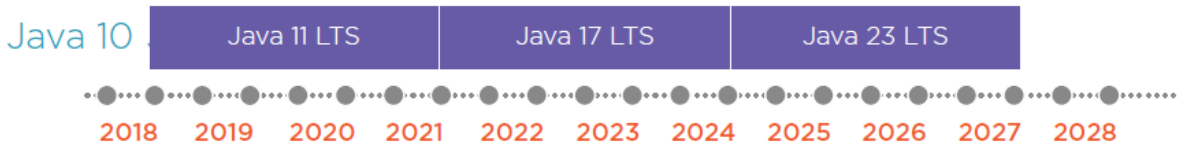
Oracle JDK 11	=	Open 11
Binary Code License		GPL v2
Paid		Free
LTS Support by Java SE Subscription		**LTS Support** by Amazon Corretto, AtoptOpen JDK (Prebuilt OpenJDK), RedHat Open JDK ..

Difference is on licensing and support (security patches and consultation).
Oracle has Oracle Binary Code License Agreement whereas OpenJDK has GPL 2. Both were free upon Java 10.

New Features in JAVA 11

Features

181: Nest-Based Access Control
309: Dynamic Class-File Constants
315: Improve Aarch64 Intrinsics
318: Epsilon: A No-Op Garbage Collector
320: Remove the Java EE and CORBA Modules
321: HTTP Client (Standard)
323: Local-Variable Syntax for Lambda Parameters
324: Key Agreement with Curve25519 and Curve448
327: Unicode 10
328: Flight Recorder
329: ChaCha20 and Poly1305 Cryptographic Algorithms
330: Launch Single-File Source-Code Programs
331: Low-Overhead Heap Profiling
332: Transport Layer Security (TLS) 1.3
333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)
335: Deprecate the Nashorn JavaScript Engine
336: Deprecate the Pack200 Tools and API

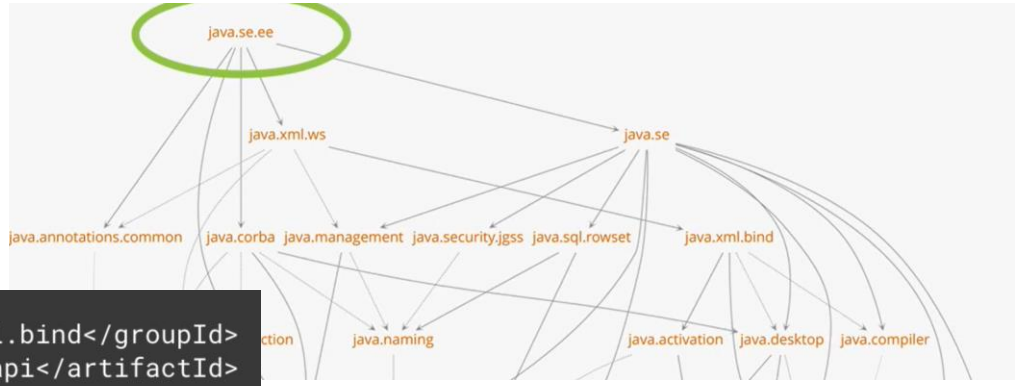


https://www.oracle.com/java/technologies/java-se-support-roadmap.html

Source code: https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/in/java11

Deprecations and removed Technologies in JAVA 11

read more client-roadmap, roadmap-update



- ❑ Applets removed: Only Java 8 Commercial Support left
- ❑ Java Web Start removed: Alternatives use tools jlink or jpackager?
- ❑ Deprecate the Pack200 Tools and API

```
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
</dependency>
```

- ❑ Nashorn deprecated (in Java 15 will be removed): Alternatives Graal.js, or Project Detroit (Google V8) – aims to be the base for a native implementation of **javax.script** package based on the Open Source JavaScript engine V8.

GraalVM provides an ECMAScript-compliant runtime to execute JavaScript and Node.js applications.

- ❑ Removed enterprise API - java.xml.*, bind(JAXB), ws(JAX-WS), ws.annotation, java.corba(CORBA), java.transaction(JTA), Java Beans Activation, wsgen, wsimport, schemagen, xjc, idlj, orbd, servertool, tnamesrv,
  Alternatives: use related maven dependencies (most of them are now Apache projects, or in App-Server Runtime)
- ❑ Removed methods: Thread#destroy(),stop(Throwable obj), java.lang.System::runFinalizersOnExit, ava.lang.Runtime::runFinalizersOnExit, SecurityManager 4 methods #**checkAwtEventQueueAccess**, and etc.

Instead of all these 4 methods you can use checkPermission(java.security.Permission)

- ❑ JavaFX – no longer part of JDK, alternative: use OpenJFX project (javafx-base, javafx-controls, javafx-media). After removing JavaFX from Java SE, Javapackager (used to create native[MSI/DMG] installers) is also removed. jpackager?

```
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx</artifactId>
    <version>11-ea+19</version>
</dependency>
```

New Features in JAVA 11

Launching Single-file

```
OLD approach:   1. >javac Hello.java   2. >java Hello
Since Java 11 (only on single file):  > java Hello.java

--source can be used to show jdk-source-level (version)
```

Scripting (exec-scripts) - Run Shebang #!

```
a.  //java --source 11 single-shebang-incmd

b. //On Win: use Cygwin or git-bash
> ./single-shebang-file
```

single-shebang-file.sh

Language and Library Improvements, Additions

- **HttpClient** (was incubator in Java 9) – replacement for HTTPURLConnection, supports http/2, WebSockets
- **Reactive Streams** – stream data support for backpressure, Flow API. Base Interface for RxJava, AKKA Streams, Spring 5
  - Flow API: Publisher, Subscription, Subscriber
  - ProcessHandle.current().destroyForcibly(); //java.lang.IllegalStateException: destroy of current process not allowed
- **String** - new methods: Unicode aware **repeat(), isBlank(), strip(), lines**
- **Files** – new methods: readString(), writeString() , encoded by UTF-8, ..
- **Optional::isEmpty (**before exists isPresent()**)**, e.g. var opt = Optional.ofNullable(null); opt.isEmpty() == true
- **Predicate::not** e.g. List.of("AB", "", "sasa").stream().filter(Predicate.not(String::isBlank)).forEach(System.out::println);

- Upgraded from **Unicode 8** to **10**: 16000 more chars, 10 new scripts, e.g. \u20BF for BitCoin
- **Local var type inference for Lambda params**: (String a, String b) -> a.concat(b) as: (a, b) -> a.concat(b) or

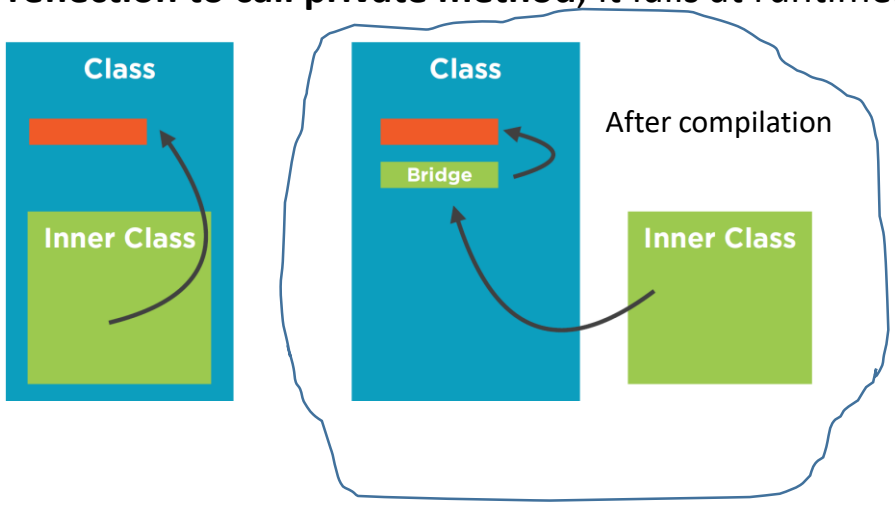
e.g. (var a, var b) -> a.concat(b), (@Nonnull var a, @Nullable var b) -> a.concat(b) //annotate types in Java

\u20BF

Limitations (no mix): (var a, b) -> a.concat(b), (var a, String b) -> a.concat(b), var a -> a.toUpperCase()

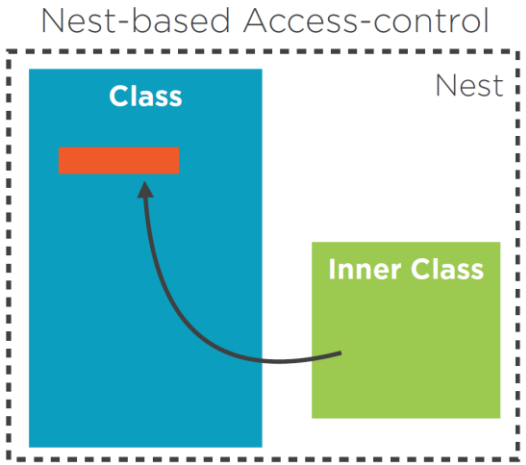
New Features in JAVA 11

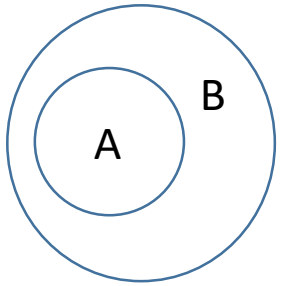
- Outer Java class which has Inner class. At Runtime inner class access private fields of Outer class via **Bridge method.** Compiler creates this **synthetic bridge method**. But once developer uses **reflection to call private method**, it fails at runtime.



Nest Based Access

- With Nest-Based Access Control mechanism inner class can access it directly in Nested manner (Nestmates). Compiler **no** need to create **bridge-method**. Inner classes can access it. This mechanism also used in **Sealed classes** in Later Java.





JEP 309 JVM Change - Dynamic Class-file Constants

Extends class-file format to support a new constant-pool form, CONSTANT_Dynamic, target language designers and compiler implementors. Loading a CONSTANT_Dynamic will delegate creation to a bootstrap method, just as linking an invokedynamic call site delegates linkage to a bootstrap method.

Improve Aarch64 Intrinsics

Optimized the existing string and array intrinsics, and implements new intrinsics for Math.sin(), Math.cos() and Match.log() on Arm64 or *Aarch64 processors*. it means better performance. *P.S An intrinsic is used to leverage CPU architecture-specific assembly code to improve the performance.*

Performance and Security Improvements

- G1GC: G1 – Garbage First garbage collector (introduced in Java 6, default since Java 9 several incremental improvements up to Java 11)

- **Epsilon GC (experimental)** - A **No-Op GC**. Apps with predictable, bounded memory usage, performance[stress] testing, short-lived obj. Allocates memory but not collect any garbage (memory allocation), once the Java heap is exhausted, the JVM will shut down. Params: -XX:+UnlockExperimentalVMOptions, -XX:+**UseEpsilonGC**

- **ZGC** - A Scalable Low-Latency **Z Garbage Collector** (Experimental) [pause time under 10 ms, maybe in future 1 ms]
- No pause time increase with heap size increase, Scale to multi-terabyte heaps. **Colored Pointers** Linux/x64 only: -XX:+UnlockExperimentalVMOptions, -XX:**+UseZGC .**

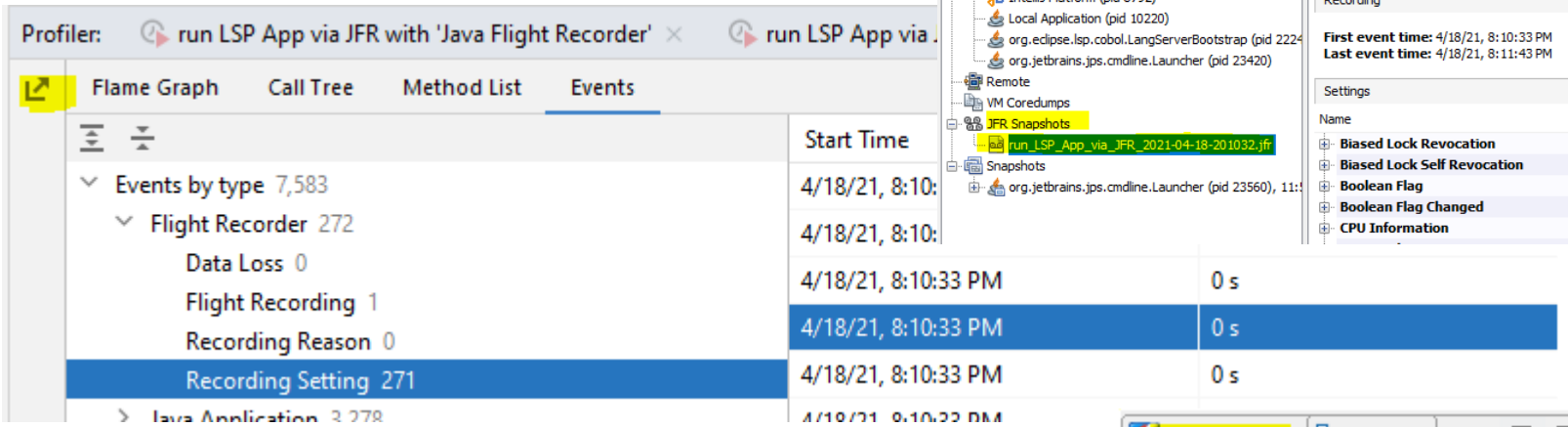
Yes ZGC provides a short pause times when cleaning up heap memories. However, it didn't return unused heap memory to the operating system, this improved in Java 13. Added support for Win in Java 14, and became prod-feature in **Java 15**

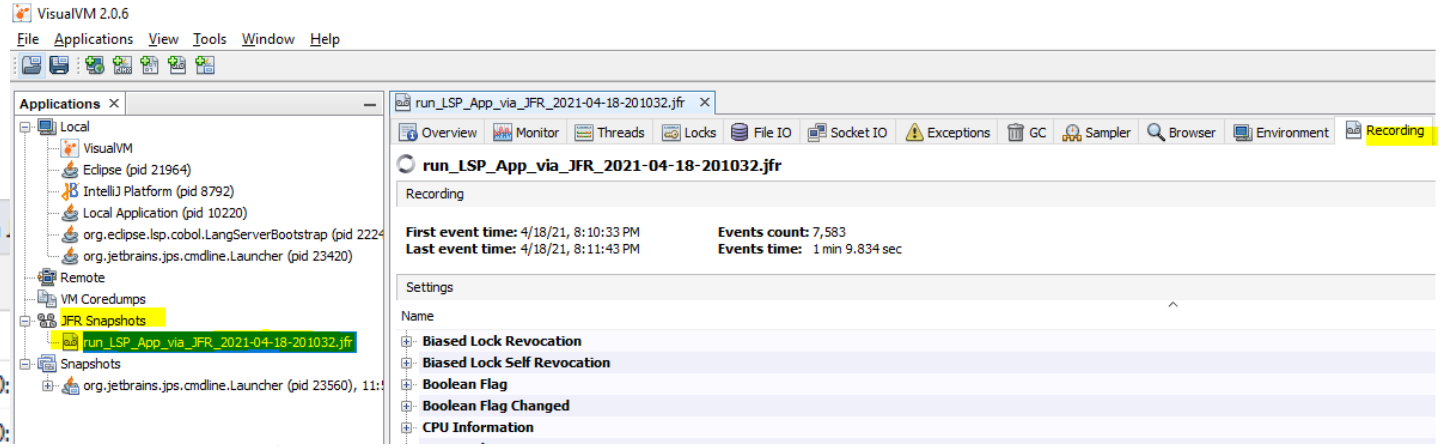
- 331: **Low-Overhead Heap Profiling**: Provide a low-overhead way of sampling Java heap allocations, accessible via JVMTI

- **TLS 1.3** – Only HTTPS, all handshake messages except first encrypted. Elliptic curve algorithm is used. Key Agreement with **Curve25519** and **Curve448**
- TLS 1.3 is not fully implemented yet, cipher mismatches, DSA certificate can not be used.
- DTLS 1.3 not implemented yet.
- ChaCha20 and Poly1305 Cryptographic Algorithms

New Features in JAVA 11

- Java Flight Recorder:
  (Telemetry events)



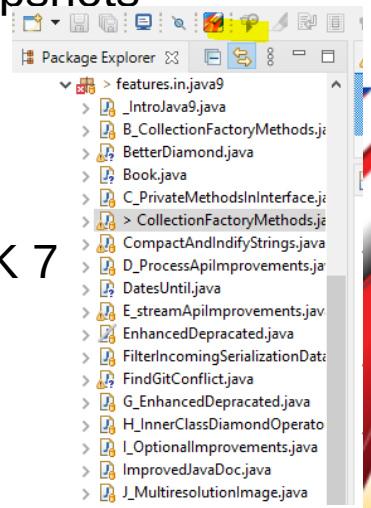


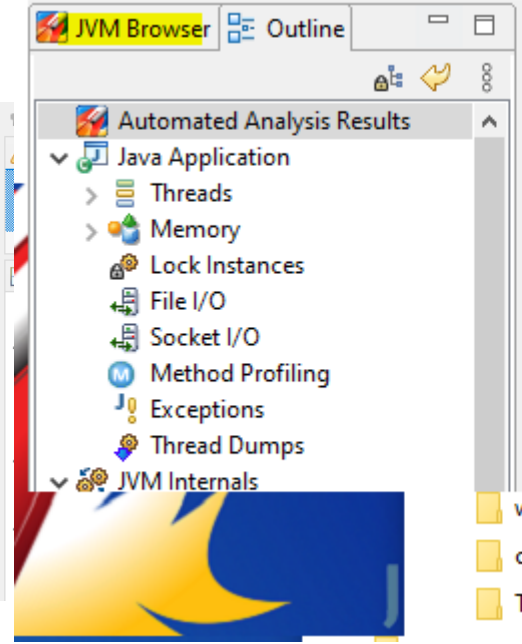
JFR is a tool for collecting diagnostic and profiling data about a running Java application.

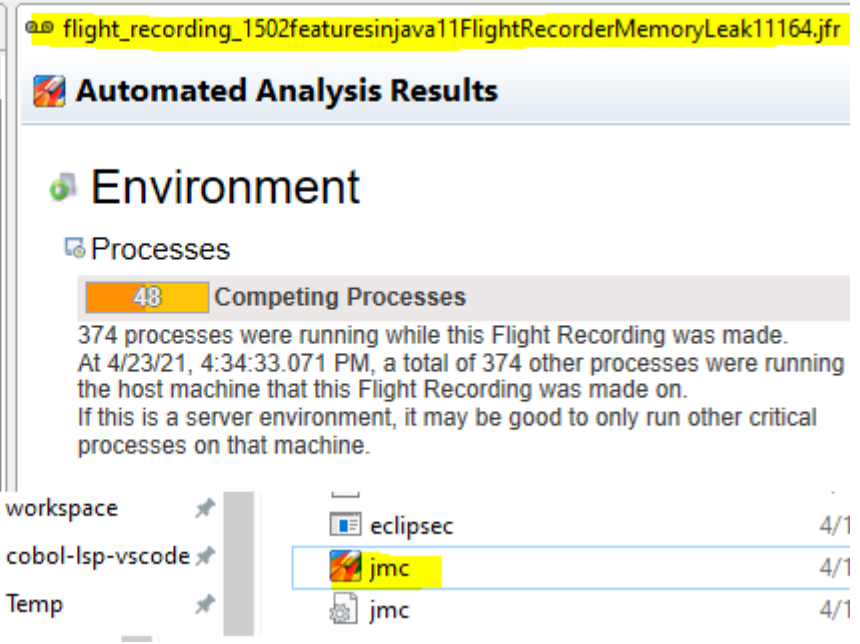
- visualVM can also see JFR snapshots
- JMC - Java Mission Control:

To see JFR output from **.jsr** file

- Supports Java Flight Recorder.
- Supports HotSpot VM, since JDK 7







- **JCMD** – Running Java process

>**jps**
>**jcmd** 11164 JFR.start duration=30s settings=profile filename=C:/workspace-JavaNew/Java-Features/leak.jfr

New Features in JAVA 12

Features

189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)

230: Microbenchmark Suite

325: Switch Expressions (Preview)

334: JVM Constants API

340: One AArch64 Port, Not Two

341: Default CDS Archives

344: Abortable Mixed Collections for G1

346: Promptly Return Unused Committed Memory from G1

Source code: https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/in/java12

New Features in JAVA 12

Language and Library Improvements, Additions

- **String::indent, ::transform** (better than inside out method call chain approach)
- **Compact Number Formats** – e.g. 1000 -> 1K, 1Mio. -> 1M . //or own **compactPatterns**

NumberFormat.getCompactNumberInstance(); NumberFormat::setRoundingMode, setMaximumFractionDigits(2)

- **Teeing Collector** - Collectors.teeing(c1, c2, combine) //apply two collectors and merge [combine function]
- **Files::mismatch** - Files.mismatch(Path.of("/file1"), Path.of("/file2")); //returns -1 if files are idendical

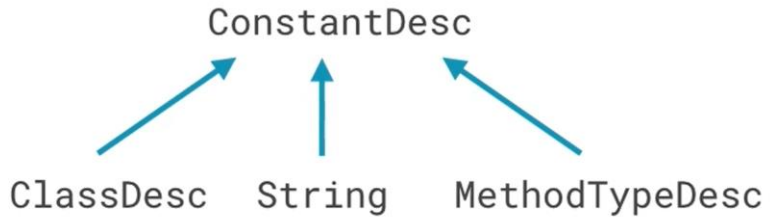
Preview Feature – Experimental (like incubator modules), Opt: **--enable-preview**

- **Switch Expressions**

- Current 'Switch Statement" - No return value, just fall through, can not define new scope (local variable)
- New Switch Expression (means value, or returns value) No fall through, return value, combine labels, block scope
- >javac [java] --enable-preview –release 12 Demo.java [Demo]
- Standardized in Java 14

```
@SuppressWarnings("preview")
public static String getDay(Day d) {
     String day = switch (d) {
     case SAT, SUN -> "Weekend";
     case MON -> "Monday";
     case TUE -> "Tuesday";
     };
     return day;
}
```

New Features in JAVA 12

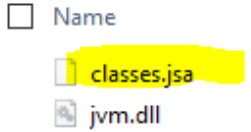


One AArch64 Port, Not Two

Before Java 12, there are two different source code or ports for the 64-bit *ARM architecture*.
- Oracle – src/hotspot/cpu/arm
- Red Hat – src/hotspot/cpu/aarch64

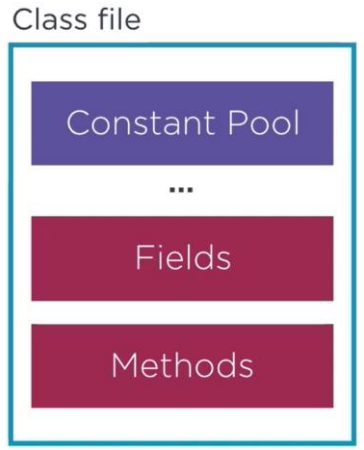
In Java 12 Oracle src/hotspot/cpu/arm port is removed, and maintain only one port src/hotspot/cpu/aarch64, and make this aarch64 the default build for the 64-bit ARM architecture.

Default CDS Archives (became standard in Java 13)



- Improved the start-up time by reuse an existing archive file.
- Before Java 12, we need to use -Xshare: dump to generate the CDS archive file for the JDK classes. Since Java 12 a new classes.jsa file in the /bin/server/ directory, a default CDS archive file for the JDK classes.

JVM Improvements



- **JEP 344**: Abortable Mixed Collections for G1: Splits the problematic collection set into two parts – mandatory and optional. The G1 will abort the optional part if lack of time to handle it. Aim is try to meet time goals.
- **JEP 346**: Promptly Return Unused Committed Memory from G1
- **New GC (experimental) – Shenandoah** [Brooks pointers] support large Heaps, Low pause times.

Contributed by RedHat to OpenJDK. Became product feature in Java 15. Oracle JDK and Open JDK has not this feature.
-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC

- JVM Constants API – **java.lang.constant** (ConstantDesc, ClassDesc, String, MethodTypeDesc)

Before issue is: No distinctions between descriptions and actual values. Now ConstantDesc to desc. const-pool entry

New Features in JAVA 12

Micro Benchmarking with JMH – JEP 230 (JDK 12 uses JMH)

Throughput – Operations per time unit / number of trx per second (**HIGHER is better**). Measure **ops/sec -** Mode.Throughput

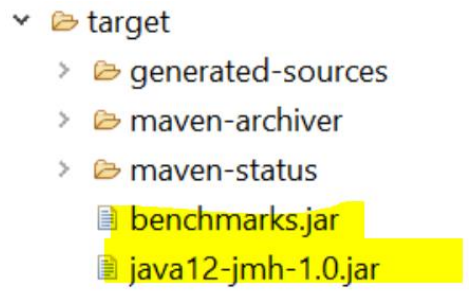
Latency [Average Time] – maximum duration of a transaction (**LOWER is better**). Measure **secs/op** - Mode..Average

- JMH is all about monitoring and measuring performance of piece of code. Measures in **nano**/micro/milli/…,
- Measure exec-time, compare alternatives algorithms, (choose algs., obj.immutability or pooling. Condition-first to avoid catching Exc)
), prevent performance regressions.
- JMH – Reproducability [repeating calls], JVM **warm-up** handling, runs benchmark multiple times, consistent reporting, multithreading support.
- @Benchmark - by default Mode.Throughput – operations per time unit, or Mode.AverageTime shows average running(exec) time or [latency] . Mode.SimpleTime (verbose Average), .. Mode.ALL
- Default: Forks 5 JVM, 5 diff. runs. Use @Fork(1) **to tune**
- TimeUnit.***NANOSECONDS , ..***
- Optional to use: @Setup, @Teardown (like in Junit)

- **Pitfalls**: Dead code eliminations (better to **return value, or Blackhole if more value to return**), compiler optimizations (combine statements, loops, constant folding), assumptions (just numbers …)
- You can integrate JMH output into **CI/CD pipeline**, to publish or visualize on different runs. Helps comparing Performance regression (another measure like regression testing)

```
//Don't Do This - via System.currentTimeMillis(), just benchmark once
//e.g. no consider optimizations, etc. .. Many runs, ..
long start = System.currentTimeMillis();
//code under benchmark
long end = System.currentTimeMillis();
long elapsed = end - start;
System.out.println("Elapsed time: " + elapsed);
```

```
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.21</version>
</dependency>
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>1.21</version>
</dependency>
```



> java -jar target/benchmarks.jar

New Features in JAVA 13

Features

350: Dynamic CDS Archives

351: ZGC: Uncommit Unused Memory

353: Reimplement the Legacy Socket API

354: Switch Expressions (Preview)

355: Text Blocks (Preview)

Java 13 All Features: https://github.com/azatsatklichov/Java-Features

Source code: https://github.com/azatsatklichov/Java-Features/tree/master/src/main/java/features/in/java13

New Features in JAVA 13

Language API Updates

- **ByteBuffer (ByteBuffer != byte[]) Improvements.**

Copy a bulk of bytes from a byte array into a specific index of the ByteBuffer.

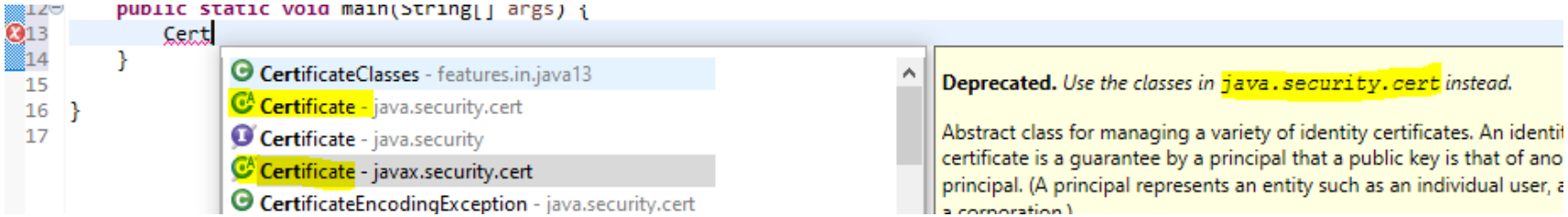
ByteBuffer put(int index, byte[] src); ByteBuffer put(int index, byte[] src, int offset, int length);

Copy a bulk of bytes from a specific index of the ByteBuffer into byte array

ByteBuffer get(int index, byte[] dst); ByteBuffer get(int index, byte[] dst, int offset, int length);

- **Removed javax.security.cert** - javax.security.cert.Certificate, javax.security.cert.X509Certificate, and exception classes

Migration to Java 13 is easy just remove **x ;) ,** use **java.security.cert**



- **Unicode 12.1 Support (from Unicode 11 to 12.1)**

- 4 new scripts (South-East Asian lang.), 555 new chars, , 61 new **emojis**, CLDR 35.1 Japanese new era, Reiwa.

Placeholder name, "NewEra", for Japanese era May 1st, 2019, replaced with the new official name Reiwa

New Features in JAVA 13

JVM Improvements

- ZGC: Uncommit Unused Memory (returns back to Main memory) **-XX:ZUncommitDelay**=<seconds> (experimental).

ZGC was introduced in Java 11 and was lack of return un-used memory but providing a short pause times (scalable low-latency: GC pause times should not exceed 10ms) when cleaning up heap memories.

Preview Feature – Experimental (like incubator modules), Opt: **--enable-preview**

- **Switch Expressions (since Java 12 still as preview feature, Standard in Java 14)**

- Current 'Switch Statement" - No return value, just fall through, can not define new scope (**local** variable)
- New Switch Expression (means value, or returns value) – No fall through, return value (yield), combine labels, block scope
- Dropped "break value" favor of added "yield" keyword to return value from switch expression
- Standardized in Java 14

```
@SuppressWarnings("preview")
public static String getDay(Day d) {
     String day = switch (d) {
     case SUN -> "Sunday";
     case MON -> "Monday";
     case TUE -> "Tuesday";
     };
     return day;
}
```

New Features in JAVA 13

`Preview Feature – Experimental  (like incubator modules), Opt:` **--enable-preview**

- **Text blocks (multi line Strings finally available ;) [before: long lines, manual escaping, indenting, … via Str+Str+Str, or StringBuffer\StringBuilder, StringWriter, Java 8 String.join, Files.lines,..])** – now at runtime works as regular str, no runtime overhead
- Normalizes line endings (always Unix style \r), trim white space[**incidental (nonessential)** ones], translate escape sequence
- String new Methods:.stripIndent(), .translateEscapes(), .formatted(). (handy on old way)
- Standardized in Java 15

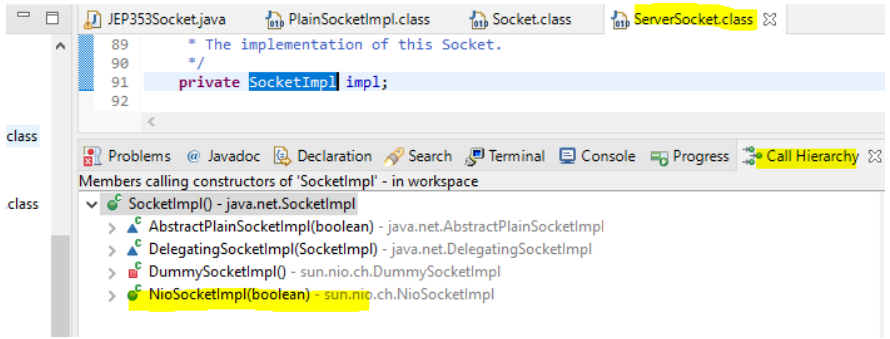
```
9       String textBlock = """
0                   Hi
1                "Hello\n\to"
2                Yes""";  //no last line
3               //"""; ///adds last line
```

Usage: a) doc REST endpoints, Swagger/OpenAPI (markdown-support) b) testing json-msg
c)Templating with formatting (for more adv. usages: Thymeleaf, Tiles, Freemarker, Velocity, ..) d) Text with escaping chars etc.

New Features in JAVA 13

Platform Changes

- **Re-implementation of Socket API** (exists since JDK 1.0, mix of legacy Java & C) - java.net.Socket, ServerSocket
- Modern and more maintainable implementation , not using stack as I/O buffer
- Made ready for new Java Concurrency models - see project Loom
- sun.nio.ch.NioSocketImpl, replacement for PlainSocketImpl, reuses NIO native code, & uses **java.concurrent.lock**
- To switch back to old PlainSocketImpl: -Djdk.net.usePlainSocketImpl
- See Java 15 for **Reimplement** the Legacy DatagramSocket API **(**java.net.DatagramSocket, MulticastSocket APIs**)**



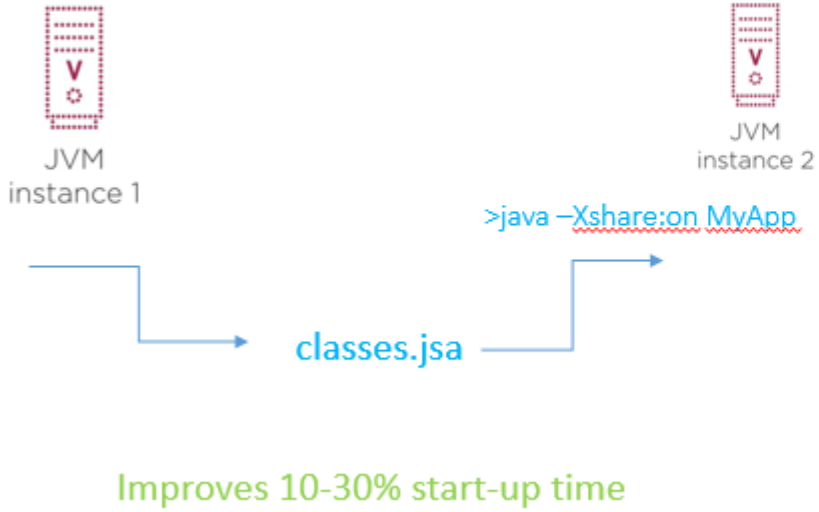
Project Loom (read more, what is wrong with async, codes like sync, works like async**)** for the purpose of supporting **easy-to-use, high-throughput lightweight concurrency** and **new programming models** on the Java platform. This is accomplished by: Virtual Threads (Fibers or AKA lightweight threads or user-mode threads). Virtual threads are lightweight threads, similar in concept to the **old Green Threads**, and are managed by the JVM rather than the kernel. The major advantages that virtual threads are intended to bring include

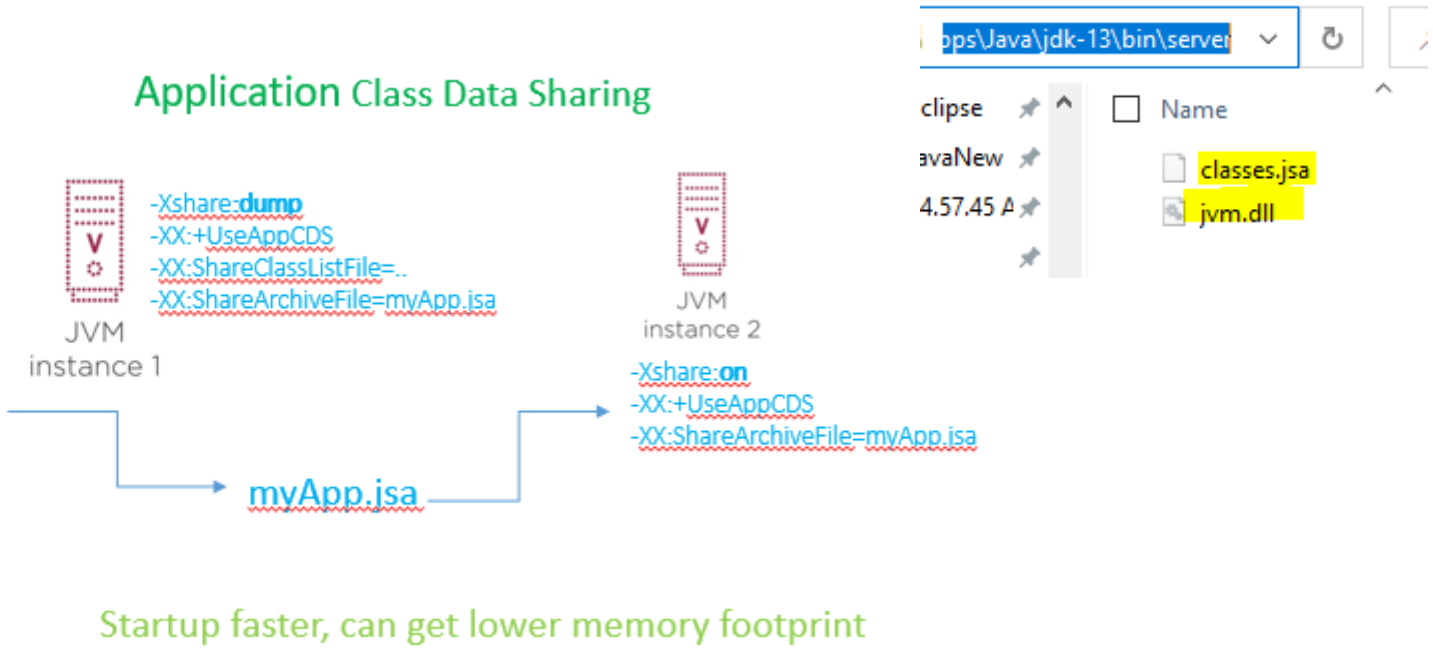
- Creating (lightweight) and blocking them is cheap, no overhead.
- Java execution schedulers (thread pools, e.g. ForkAndJoin & etc) can be used.
- There are no OS-level data structures for the stack.

New Features in JAVA 13

Class Data Sharing – ACDS (Firstly in introduced in Java 10 as preview)

- CDS introduced in Java 10, improves startup performance by creating a class-data archive once and reusing it so that the JVM needs not to recreate it again. The primary motivation for including CDS in Java SE is to decrease in startup time
- ACDS added in Java 13. Also supports classes in custom class loaders
- This command creates Dynamic CDS archive file: java -XX:ArchiveClassesAtExit=hello.jsa -cp hello.jar Hello
- This command runs a .jar with an existing CDS archive file: bin/java -XX:SharedArchiveFile=hello.jsa -cp hello.jar Hello
- **Deprecated flags** (disable class verification)**:** -Xverify:none, -noverify
- Use AppCDS instead for upfront verification, because they are already pre-verified (generated on app exit), + performant
- For more details here





Miscellaneous Changes

Apart from the JEPs listed above, Java 13 has given us a few more notable changes:

- java.nio – method FileSystems.newFileSystem(Path, Map<String, ?>) added
- javax.crypto – support for MS Cryptography Next Generation (CNG)
- javax.security – property jdk.sasl.disabledMechanisms added to disable SASL mechanisms
- javax.xml.crypto – new String constants introduced to represent Canonical XML 1.1 URIs
- javax.xml.parsers – new methods added to instantiate DOM and SAX factories with namespaces support
- Support added for Kerberos principal name canonicalization and cross-realm referrals

THANK YOU

References

https://www.baeldung.com/oracle-jdk-vs-openjdk
https://medium.com/@javachampions/java-is-still-free-c02aef8c9e04
https://www.journaldev.com/13106/java-9-modules
https://mkyong.com/java/what-is-new-in-java-15/
https://blogs.oracle.com/javamagazine/post/java-project-amber-lambda-loom-panama-valhalla
https://docs.oracle.com/en/java/javase/17/language/java-language-changes.html#GUID-67ED83E7-D79F-4F46-AA33-41031E5CD094