Java 9, 10, 11

Dokumentace

- http://openjdk.java.net/projects/jdk9/
- http://openjdk.java.net/projects/jdk/10/
- http://openjdk.java.net/projects/jdk/11/

Java Module System (jigsaw)

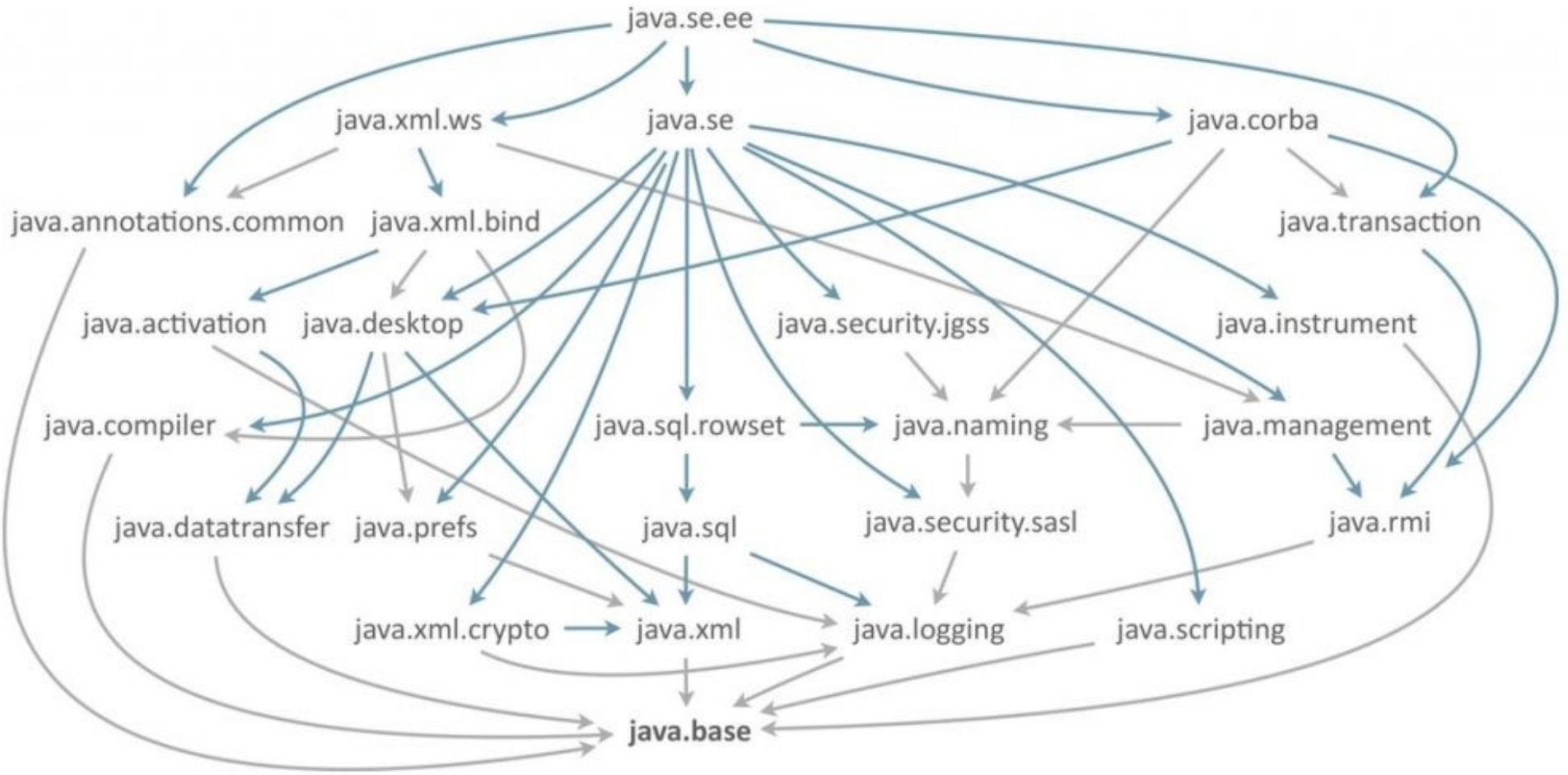
- Proč moduly? Aplikace se skládá z řady spolu interagujících částí (často reprezentovaná jednotlivými JAR soubory). Dají se z toho dělat i pěkné grafy a je přehledně vidět co na čem závisí … jakmile se ale aplikace spustí, vytvoří se classpath, ve které je jenom plochá struktura balíčků a vztahy mezi jednotlivými částmi aplikace zmizí.
- Java Module System má za cíl tuto situaci změnit a přidat do runtime prostředí strukturovaný model aplikace. První, kdo tento nový modulární systém využívá je JRE.

rt.jar → jmods

- V Java 8 je soubor rt.jar, kde se nachází všechny zkompilované Java třídy, které jsou součástí veřejného API (třídy z balíčků java, javax, org), ale i neveřejné API Javy, které není doporučené používat (třídy z balíčků com, jdk, sun) – v Java 8 má kolem 60MB.
- V Java 9 soubor rt.jar není a místo toho je tam adresář jmods se soubory s příponou jmod.
- V současnosti je JMOD formát postaven na ZIP formátu.

https://stackoverflow.com/questions/44732915/why-did-java-9-introduce-the-jmod-file-format

Java Platform Modules



- Komplexní graf závislostí:
  - http://www.baeldung.com/wp-content/uploads/2017/03/jdk-tr1.png

Zpětná (ne)kompatibilita

- Třídy ze sun.misc balíčku jsou interní a není je možné používat.
  - S výjimkou sun.misc.Unsafe:
  - http://gregluck.com/blog/archives/2017/03/using-sun-misc-unsafe-in-java-9/
- Následující moduly nejsou ve výchozím nastavení součástí classpath (a v budoucnosti budou z Java SE odstraněny!!!):
  - java.activation
  - java.annotations.common
  - java.corba
  - java.transaction
  - java.xml.bind
  - java.xml.ws

JAXB !!!

Jak použít JAXB (nejlépe přidat dependency):

https://stackoverflow.com/questions/43574426/how-to-resolve-java-lang-noclassdeffounderror-javax-xml-bind-jaxbexception-in-j

https://blog.frankel.ch/migrating-to-java-9/1/

Zpětná (ne)kompatibilita

- Java 9 je skutečně Java 9, ne jako u Java 8, kdy interní označení bylo 1.8. Detail, ale vsadím se, že tahle drobná změna pár věcí rozbije …
- Řada interních tříd z com.sun balíčku změnila package, aby bylo možné je zapouzdřit uvnitř modulů
- Pokud potřebujete přístup k internímu Java API, pak je možné je zpřístupnit pomocí --add-exports (samozřejmě to je nedoporučované) :-)
- Endorsed mechanismus byl kompletně zrušen, to samé tools.jar
- S Java 9 kompilátorem je možné kompilovat do Java 9, 1.8, 1.7 a 1.6. RIP 1.5 :-)
- Java 9 Migration Guide:
  - https://docs.oracle.com/javase/9/migrate/toc.htm

Java 11: The Great Removal

- Již v Java 9 zmizelo:
  - Java Visual VM (nyní lze stáhnout z GitHubu)
  - 32 bit verze JDK
- V Java 11 zmizelo:
  - Desktop JRE (viz. dále)
  - Java applety, Applet Viewer, Java Web Start, browser plug-in, Java Control Panel s automatickými aktualizacemi, javaws, javapackager
  - Java EE & Corba moduly (nyní z Centralu)
  - Java Mission Control (lze stáhnout zvlášť)
  - Java FX (nyní z Centralu)
- Brzy zmizí:
  - Nashorn (JavaScript engine)
- https://news.kynosarges.org/2018/09/26/java-se-11-the-great-removal/

JAXB, JAX-WS

- Knihovna JAXB byla v Java 11 odstraněna:
  - https://jaxenter.com/jdk-11-java-ee-modules-140674.html
- Poslední verze JAXB fungují s Java 11, stačí přidat tuto dependency:
  - https://javalibs.com/artifact/org.glassfish.jaxb/jaxb-runtime
- Knihovna JAX-WS byla v Java 11 odstraněna, to se týká také konzolových nástrojů wsgen & wsimport:
  - http://openjdk.java.net/jeps/320

Class path vs. Module path

- V Java 9 máte na výběr:
  - Použít tradiční class path
  - Nebo nově module path
- Class path je v Java 9 ve skutečnosti jeden velký nepojmenovaný modul (unnamed module), který exportuje všechny balíčky a requires všechny knihovny.
  - http://www.logicbig.com/tutorials/core-java-tutorial/modules/unnamed-modules/

module-info.java

- Nový typ třídy, která se musí nacházet v defaultním balíčku.
  Obsahuje dvě sekce: requires a exports.
  - requires: Na jakých modulech aktuální modul závisí
  - exports: Jaké balíčky jsou veřejným API tohoto modulu
    - Poznámka: Všechny balíčky, které nejsou uvedené v exports, nejsou jiným modulem viditelné!!! A exportování nefunguje transitivně!!!

public != visible
public + exports = visible

```
module cz.jiripinkas.hello {

    requires log4j.api;

    exports cz.jiripinkas.hello;

}
```

Poznámka I: K tomu, abyste zjistili, na čem nějaká knihovna závisí existuje nová aplikace jdeps.
Použití: `jdeps -s NAZEV.jar`
Nejvíc se mi ale zatím osvědčilo otevřít module-info.class v IntelliJ Idea
(která provede dekompilaci)
Poznámka II: module, requires & exports NEJSOU klíčová slova

Module hell?

- Každý Java vývojář dříve nebo později narazí na JAR hell:
  - https://dzone.com/articles/what-is-jar-hell
- Čeká nás to samé s moduly? To ukáže až čas, ale ten největší problém (konflikty verzí stejné knihovny) moduly neřeší. A v současnosti přidávají kupu dalších …
  - https://blog.codefx.org/java/dev/will-there-be-module-hell/

Pojmenování modulů I.

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

- Vlastní pojmenování modulů:
  - Název modulu by se měl odvozovat od názvu hlavního balíčku:
    - http://blog.joda.org/2017/04/java-se-9-jpms-modules-are-not-artifacts.html

Pojmenování modulů II.

- Automatické moduly:
  - http://blog.joda.org/2017/05/java-se-9-jpms-automatic-modules.html
  - Když Vaše aplikace používá nějakou dependency X a obsahuje module-info.class, pak také musí dovnitř přidat require X; Co je ale to X?
    - Pokud dependency X obsahuje module-info.class, pak je název modulu uvnitř.
    - Pokud neobsahuje module-info.class, pak může být v MANIFEST.MF klíč s názvem „Automatic-Module-Name“.
    - Pokud ani toto není, pak je název modulu roven názvu JAR souboru. Navíc v názvu modulu není povolené „mínus“, tudíž například pro dependency s artifactId „camel-core“ je název modulu „camel.core“.

Split Packages Problem I.

- Java 9 moduly mají jedno významné omezení:
  - Dva moduly nesmí obsahovat stejný package. Jedná se o výrazné architektonické omezení, které rozbije hodně stávajících aplikací.
    - https://stackoverflow.com/questions/42358084/package-conflicts-with-automatic-modules-in-java-9
- Příklad:
  - Log4j2 2.7 (naštěstí aktuálně rok stará verze) na Java 9 rozhodně nebude fungovat (a stejný problém má hromada knihoven, které jsou rozčleněné do mnoha na sobě závisejících artifactů) … SLF4J, Spring, Hibernate, Lucene, …

Split Packages Problem II.

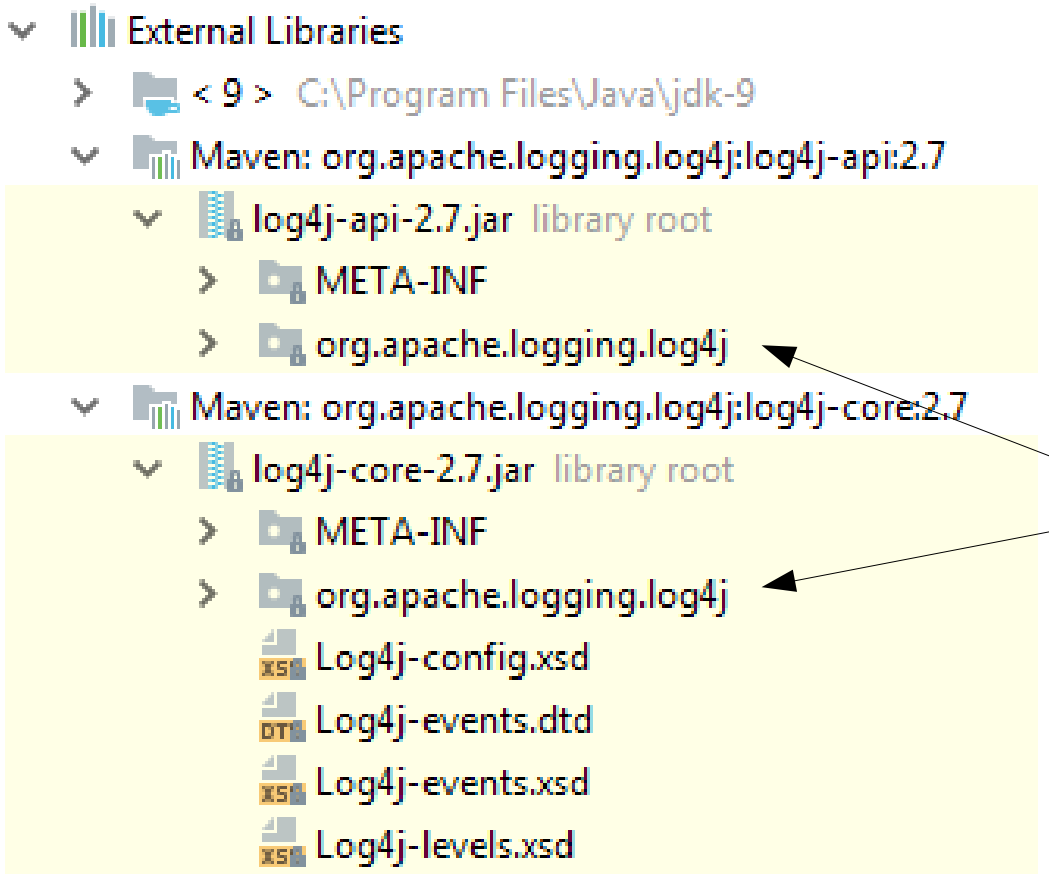
1. dependency:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.7</version>
</dependency>
```

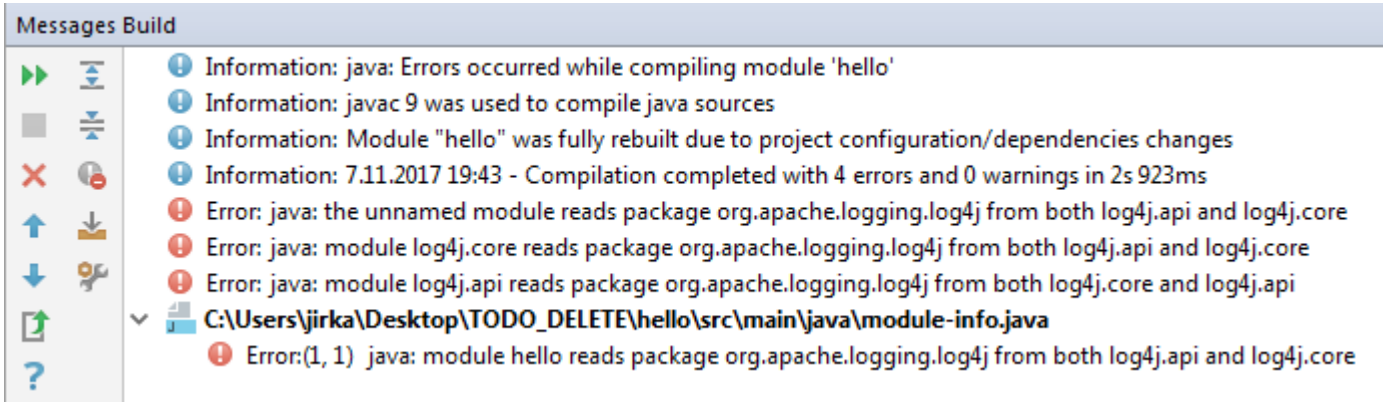
2. module-info.java:

```
module hello {
    requires log4j.api;
}
```

Split Packages Problem III.



Stejně pojmenované hlavní balíčky



Split Packages Problem IV.

3. Main třída:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
public class TestLog4j2 {
    private static final Logger log = LogManager.getLogger(TestLog4j2.class);
    public static void main(String[] args) {
        log.info("Logging Works ^_^");
    }
}
```

Split Packages Problem V.

4. log4j2.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="warn">
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </Console>
    </Appenders>
    <Loggers>
        <Root level="info">
            <AppenderRef ref="Console"/>
        </Root>
    </Loggers>
</Configuration>
```

Java 9 & Reflexe

- U reflexe máme dva způsoby přístupů:
  - Klasické public atributy / metody
  - Po zavolání metody setAccessible(true) se dají zpřístupnit i private atributy / metody („deep reflection“)
- V modulárním světě s prvním přístupem není problém, balíček s takovou třídou jenom musí být zpřístupněn pomocí „exports“.
- U deep reflection je jiná situace, balíček navíc musí být otevřen pomocí „opens“. Máme několik možností:
  - `opens nazev.balicku;`
  - `opens nazev.balicku to nazev.modulu;`
  - `open module { … }`
  - http://in.relation.to/2017/04/11/accessing-private-state-of-java-9-modules/

Illegal Reflective Access

- V Java 9 jsem snad v každé aplikaci narazil na tento warning (při použití classpath, u module path je chyba):

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by com.sun.xml.bind.v2.runtime.reflect
WARNING: Please consider reporting this to the maintainers of com.sun.xml
WARNING: Use --illegal-access=warn to enable warnings of further illegal
WARNING: All illegal access operations will be denied in a future release
```

- To znamená, že se pomocí reflexe používá nějaký private atribut / metoda z balíčku v JRE, který není otevřen pro „hlubokou reflexi“ (a pomocí --illegal-access=warn se aktivují warning výpisy pro všechny takové přístupy).
- Příklad (v současnosti v ovladači pro PostgreSQL):

```
Field defaultTimeZone = TimeZone.class.getDeclaredField("defaultTimeZone");

defaultTimeZone.setAccessible(true);

defaultTimeZone.get(TimeZone.getDefault());
```

- https://medium.com/codefx-weekly/reflective-access-maven-on-java-9-and-speeding-through-the-night-a71ab23b6364

Reálné použití modulů

- Jak hodně knihoven je připravených na reálné použití v module path?
  - https://javalibs.com/charts/java9

Cyclic dependency

- Cyklické závislosti mezi moduly jsou zakázané. Naštěstí Maven toto také nepovoluje a obecně to nikdy nebyl best practice, tak snad se toho tolik nerozbije :-)

jlink

- Java 9 obsahuje nový experimentální nástroj, který umožňuje vytvořit stand-alone Java aplikaci s custom JVM:
- Linux:

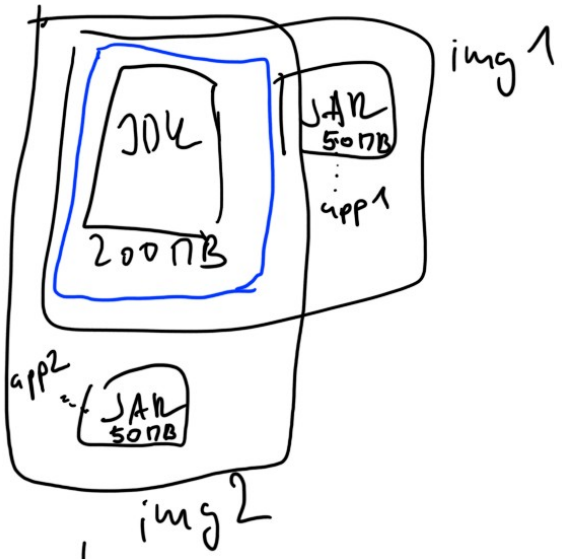
```
jlink --module-path $JAVA_HOME/jmods:target/hello-1.0-SNAPSHOT.jar --
add-modules cz.jiripinkas.hello --launcher
cz.jiripinkas.hello=cz.jiripinkas.hello/main.Main --output my-app --
strip-debug --compress 2 --no-header-files --no-man-pages
```

- Windows:

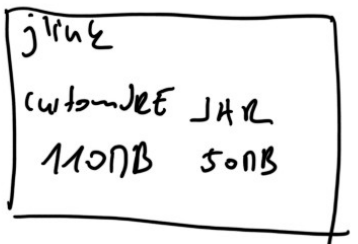
```
jlink --module-path "%JAVA_HOME%\jmods;target\hello-1.0-SNAPSHOT.jar"
--add-modules cz.jiripinkas.hello --launcher
cz.jiripinkas.hello=cz.jiripinkas.hello/main.Main --output my-app --
strip-debug --compress 2 --no-header-files --no-man-pages
```

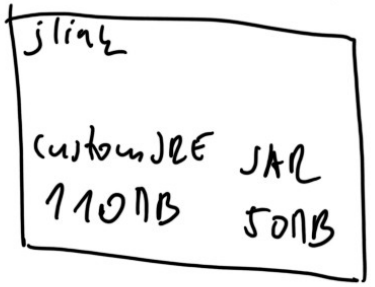
  - https://steveperkins.com/using-java-9-modularization-to-ship-zero-dependency-native-apps/
  - Poznámka: jlink build je OS-specific!

Velikost několika custom JRE s jlinkem může být klidně větší než několik aplikací s JRE / JDK a Dockerem



VS.





200MB + 50MB + 50MB = 300MB

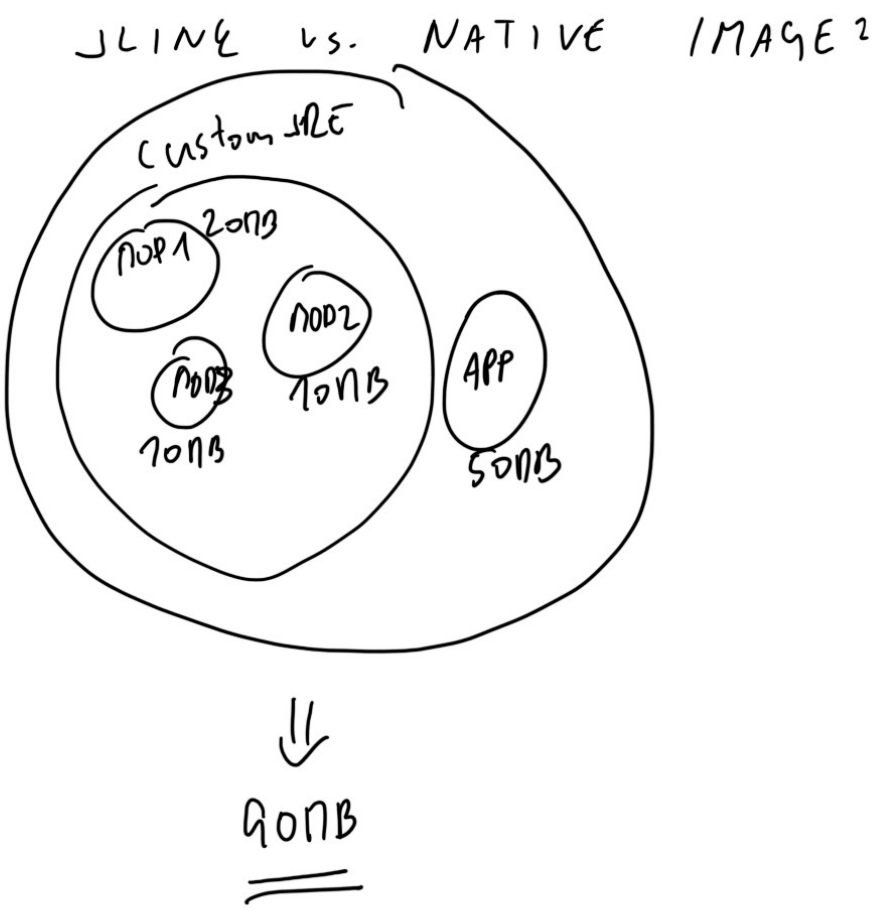
AdoptOpenJDK JDK → "JRE"
200 40

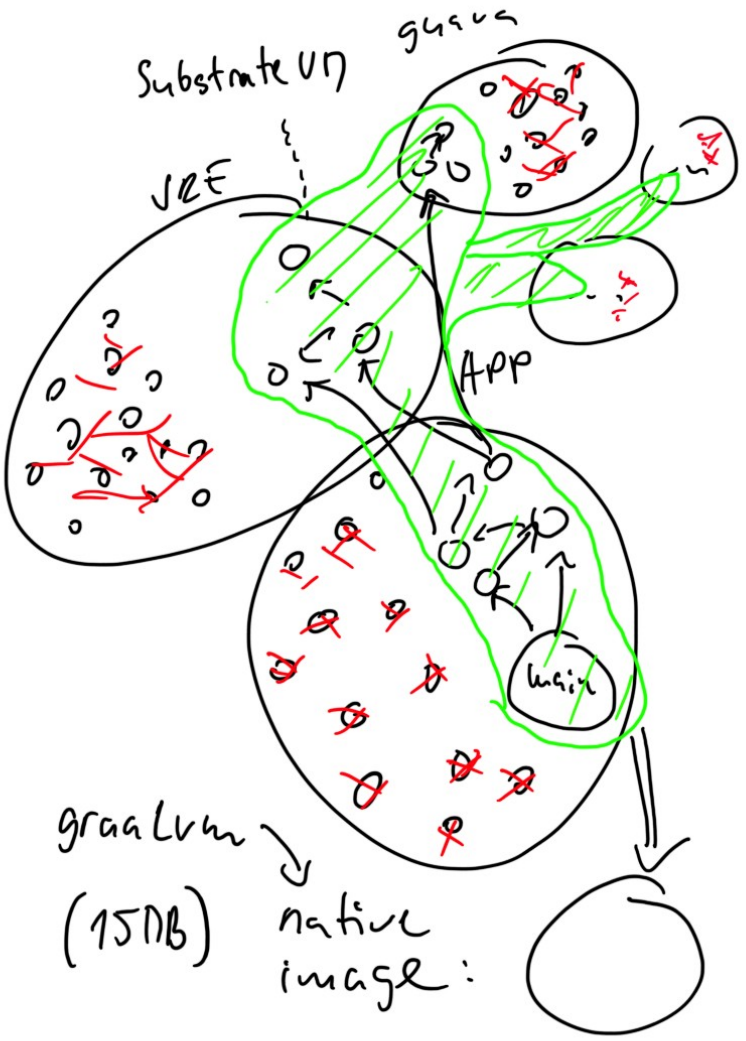
40MB + 50 + 50 = 140MB

110 + 50 + 110 + 50 = 320MB

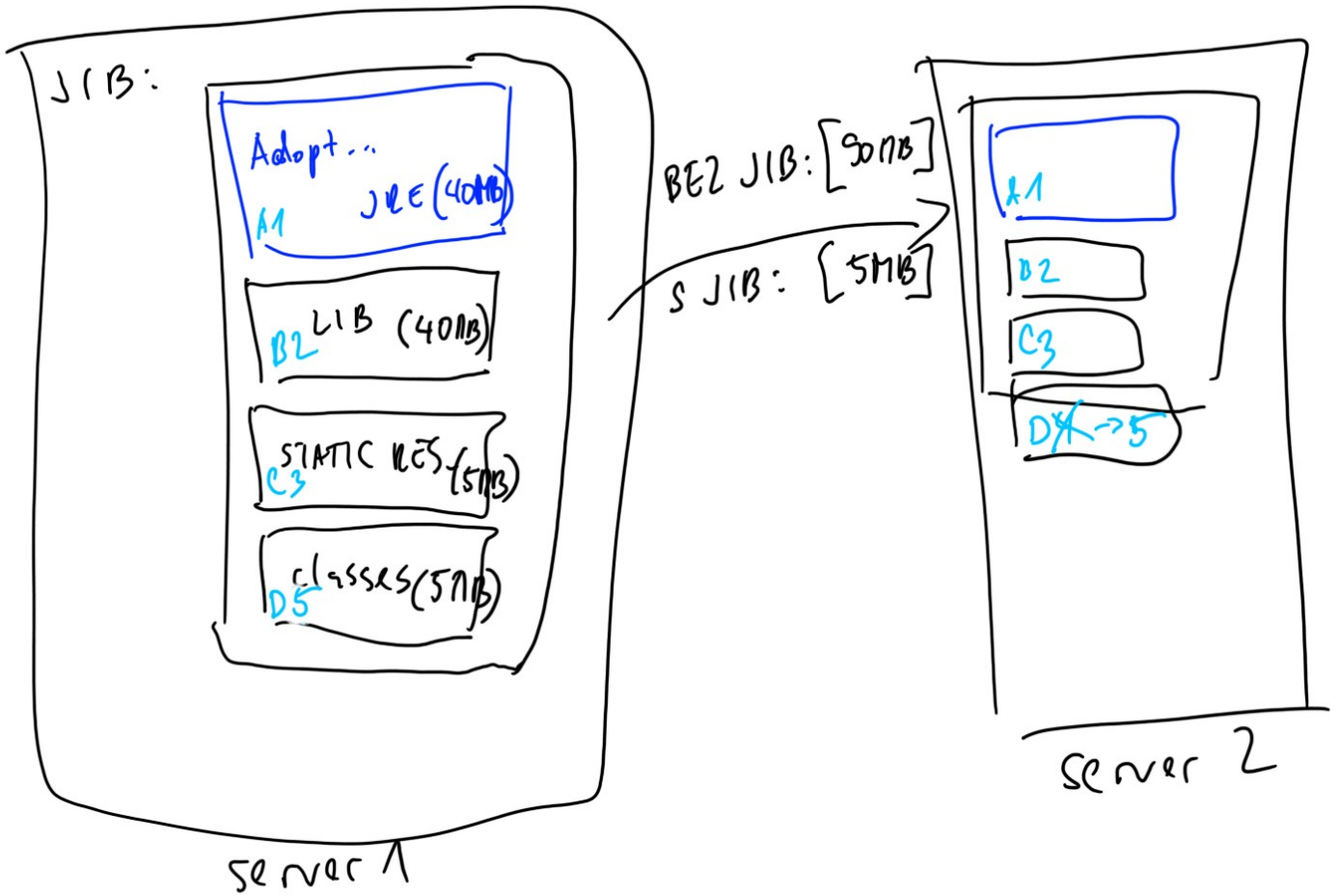
JLINK

jlink vs. Native Image





Optimalizace přenosu dat s JIBem (nebo obecně se správným vrstvením Docker image):



Spring Boot

- Spring Boot 2.1 je kompatibilní s Java 11.
- Spring je od verze 5 kompatibilní s module path, ale různé další často používané knihovny (jako Hibernate) dosud není doporučené používat s module path.

Vylepšení Infinite Streamů

- Nové intermediate metody: takeWhile(), dropWhile():

```
Stream.generate(() -> new Random().nextInt(1_000))
    .takeWhile(i -> !i.equals(0))
    .forEach(System.out::println);
```

  - https://blog.codefx.org/java/java-9-stream/

Vylepšení Optional

- Nová metoda ifPresentOrElse():

```
Optional<Object> optional = Optional.empty();
optional.ifPresentOrElse(o -> {
    System.out.println("value exists: " + o);
}, () -> {
    System.out.println("value doesn't exist");
});
```

- Další metody: stream(), or(), orElseThrow(), isEmpty()
  - https://blog.codefx.org/java/java-9-optional/

Vylepšení kolekcí

- Map.ofEntries():

```
Map.ofEntries(
    Map.entry("key1", "value1"),
    Map.entry("key2", "value2"),
    Map.entry("key3", "value3")
);
```

- Set.of():

```
Set.of("a", "b", "c");
```

- List.of():

```
List.of("a", "b", "c");
```

- List.copyOf():

```
var copy = List.copyOf(list)
```

Try-with-resources vylepšení

- Před Java 9 pouze:

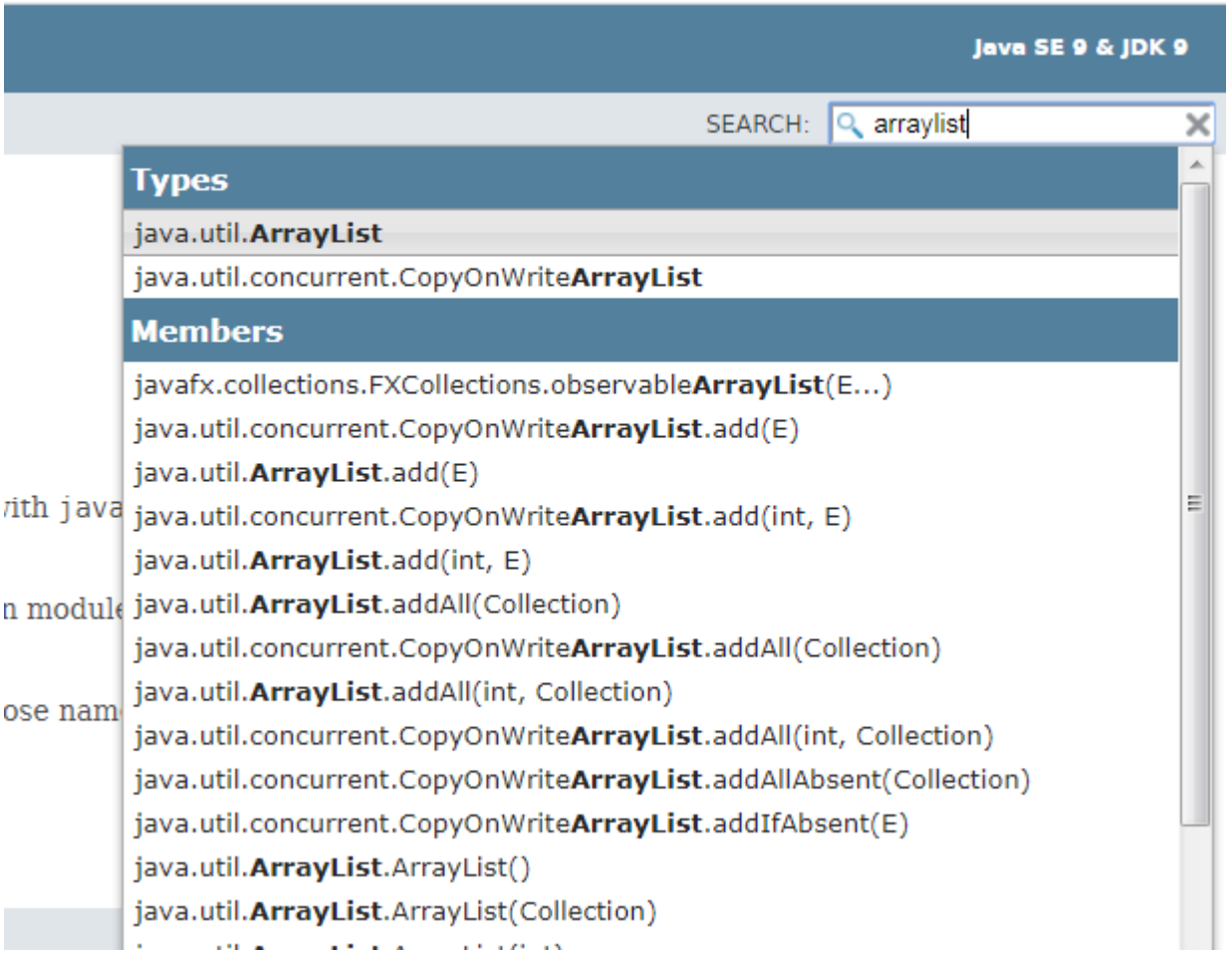
```
try(Reader reader = Files.newBufferedReader(Paths.get("pom.xml"))) {
    // do stuff
}
```

- Od Java 9 také funguje (proměnná musí být final nebo effectively-final):

```
Reader reader = Files.newBufferedReader(Paths.get("pom.xml"));
try(reader) {
    // do stuff
}
```

Vylepšení Javadocu

- HTML Javadoc KONEČNĚ obsahuje políčko pro vyhledávání:
- Poznámka: samozřejmě to funguje nejenom v Javadocu pro JRE, ale i pro Vaše projekty :-)



https://docs.oracle.com/javase/9/docs/api/overview-summary.html

Private static & instance methods on interfaces

- V Java 8 byly uvedeny default & statické metody, které je možné deklarovat v interface. Mělo to ale drobné omezení … tyto metody byly pouze public. V Java 9 nyní můžete používat private statické a instanční metody.

Deprecations

- Applet API, Corba, Observer & Observable
- `new Integer(10)` apod. je (konečně!) deprecated!!! Místo toho se používají metody jako valueOf(), parse…()
- Anotace @Deprecated doznala užitečného vylepšení:

```
@Deprecated(since = "forever", forRemoval = true)
```

jshell

- Java má nyní REPL (jshell)
  - https://en.wikipedia.org/wiki/Read–eval–print_loop

G1

- Výchozí Garbage Collector je od Java 9: G1. Nachází se v Javě už od Java 8, ale v té byl výchozí Parallel GC.
  - http://www.oracle.com/technetwork/tutorials/tutorials-1876574.html
- Tento Garbage Collector má potenciálně velice užitečnou funkci String Deduplication:
  - https://stackoverflow.com/questions/27949213/string-deduplication-feature-of-java-8
- Concurrent Mark Sweep (CMS) GC je deprecated.
  - http://openjdk.java.net/jeps/291

Další novinky

- Stack walking API:
  - https://www.javaworld.com/article/3188289/core-java/java-9s-other-new-enhancements-part-5-stack-walking-api.html
- Compact Strings (Strings density project) – automaticky zapnuté – znaky Stringů, které obsahují pouze LATIN-1 (ISO 8859-1) znaky, budou reprezentované jedním bytem, ostatní budou reprezentované dvěma byty (UTF-16) – drobnost, která by ale měla zvýšit rychlost Javy a snížit množství používané paměti:
  - https://www.javagists.com/compact-strings-java-9

Další novinky

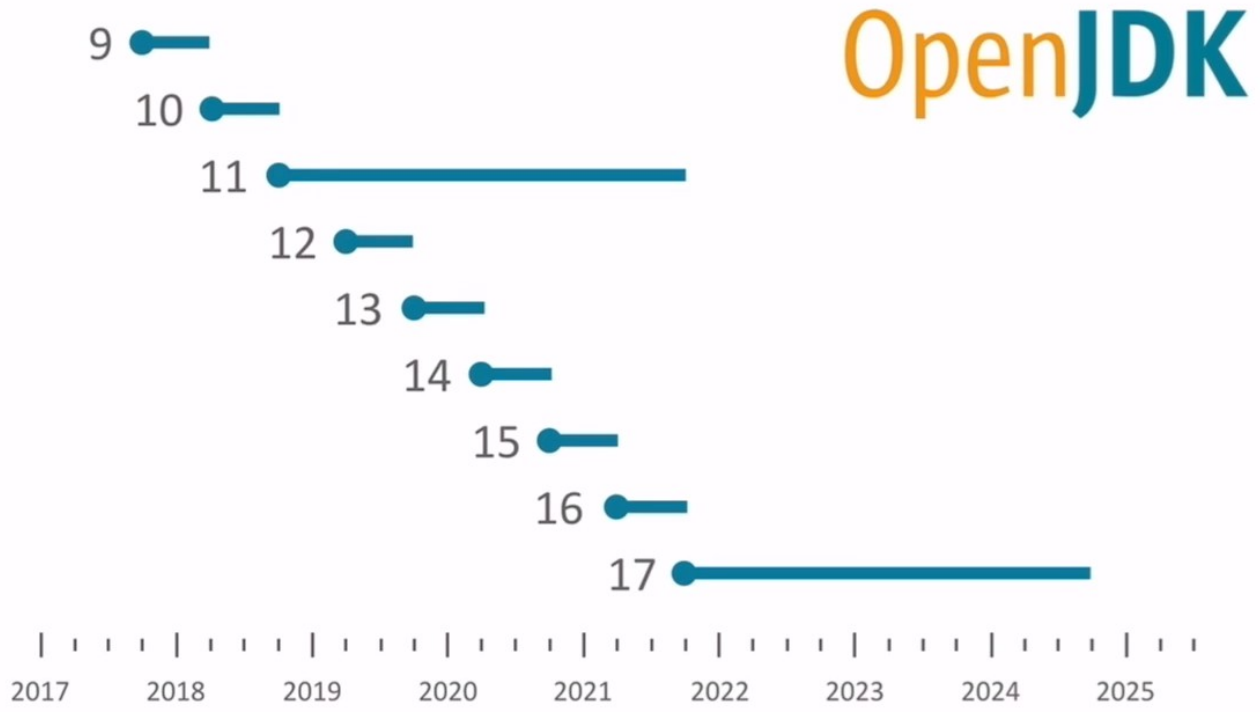
- Logování JDK tříd pomocí Log4J nebo Logback:
  - https://github.com/CodeFX-org/demo-java-9/tree/master/src/org/codefx/demo/java9/api/platform_logging
- Reactive Streams:
  - http://www.baeldung.com/java-9-reactive-streams
- HTTP 2 client (experimental):
  - https://dzone.com/articles/java-9-http-20

Další novinky

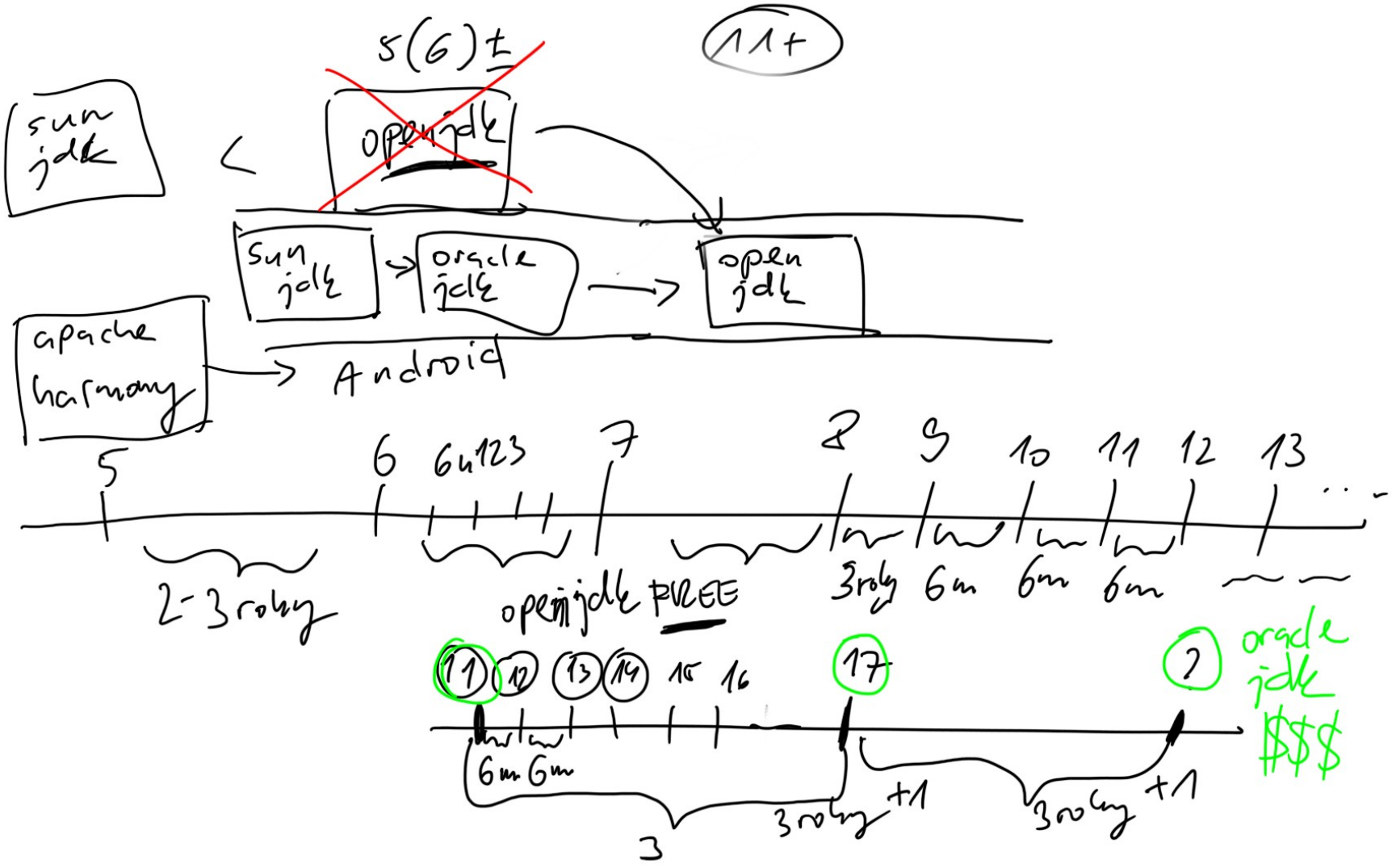
- VisualVM už není součástí JDK (mimo jiné hprof a jhat nástroje byly zrušeny bez náhrady):
  - https://visualvm.github.io/
- Flight Recorder (součást Java Mission Control) bude v blíže neupřesněné budoucnosti plně open-source:
  - https://www.reddit.com/r/java/comments/6yh4yq/oracle_will_also_open_source_commercial_features/
- Pár zbylých novinek je zde:
  - http://openjdk.java.net/jeps/291

Java Roadmap >= 10

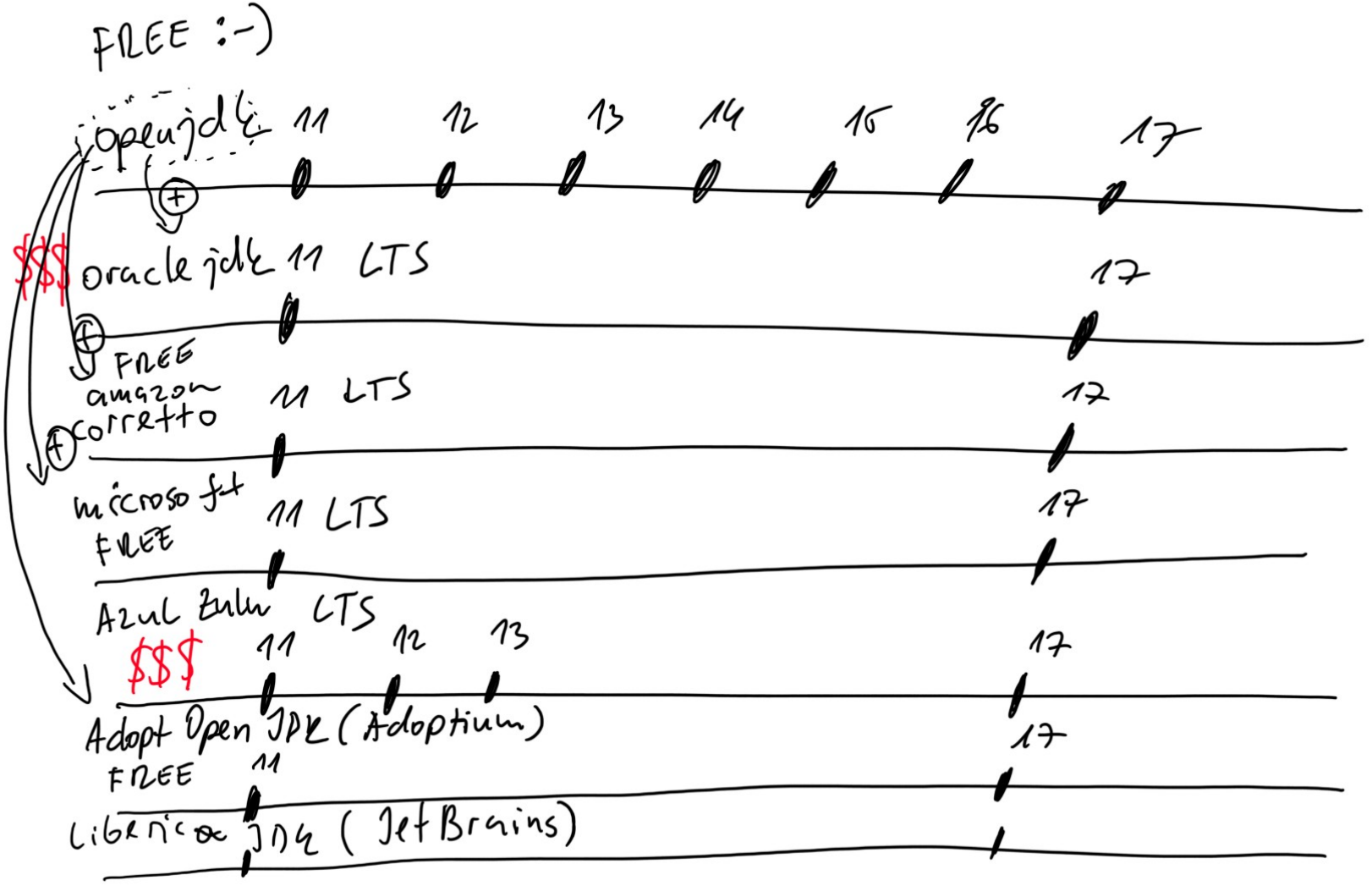
- Java 10 se nakonec bude jmenovat Java 10, přemýšlelo se, že se bude jmenovat Java 18.3 (YY.M), ale nakonec se od toho upustilo. Každé 3 roky bude LTS (Long Term Support) release. První LTS release bude Java 11 (POZOR! Java 9 NENÍ LTS release, veřejné aktualizace skončí vydáním verze Java 10!!!):



Sun JDK → Oracle JDK → Open JDK



Flavours of Open JDK



Multi-Release JAR

- https://blog.codefx.org/tools/multi-release-jars-multiple-java-versions/

Vylepšení integrace s Docker

- Poznámka: Je hodně dobrý nápad používat Docker minimálně s JDK 10 (resp. JDK 11 :-))
  - https://blog.csanchez.org/2017/05/31/running-a-jvm-in-a-container-without-getting-killed/
  - https://blog.csanchez.org/2018/06/21/running-a-jvm-in-a-container-without-getting-killed-ii/
- Při spuštění Docker containeru je možné specifikovat maximální velikost alokované paměti pomocí `--memory 1000m`
- Java automaticky přidělí heapu 1/4 této velikosti (256MB)
- Množství alokované paměti pro heap se dá ovlivnit pomocí přepínačů

  `-XX:MaxRAMPercentage=50` (toto nastaví velikost heapu na cca. 50% přidělené paměti, čili cca. 500MB)

  další přepínače: `-XX:MinRAMPercentage, -XX:InitialRAMPercentage`

Vylepšení Stringu

- Nové metody (od Java 11):
  - isBlank(), lines(), strip(), stripLeading(), stripTrailing(), repeat():
  - https://www.journaldev.com/26288/java-11-new-methods-in-string-class

Vylepšení ResourceBundle

- ResourceBundle od Java 9 načítá properties v UTF-8 na místo ISO-8859-1
- https://openjdk.java.net/jeps/226

var (od Java 10)

- Od Java 10 je nové klíčové slovo: var.
  - Místo:

```
String text = "Hello Java 9";
```

  - Můžete nyní použít:

```
var text = "Hello Java 10";
```

  - Jedná se jenom o „syntactic sugar“, kompilátor nahradí „var“ příslušným typem (je to přímo vidět v class souborech).
  - Osobně nejsem velkým příznivcem této nové funkcionality, ale může to v některých situacích pomoci k čitelnějšímu kódu:
    - http://openjdk.java.net/projects/amber/LVTIstyle.html

HttpClient

- V Java 9 byl uveřejněn nový HTTP client (experimentální), v Java 11 byl finalizován:
  - Nový HTTP client umí jak synchronní, tak asynchronní požadavky, umí pracovat jak s HTTP/1.1, tak s HTTP/2 a má hezké API:

```
var request = HttpRequest.newBuilder()
    .uri(URI.create("https://javalibs.com"))
    .build();
var client = HttpClient.newHttpClient();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

  - https://winterbe.com/posts/2018/09/24/java-11-tutorial/
  - https://blog.codefx.org/java/reactive-http-2-requests-responses/

InputStream transferTo()

- Od Java 9 je nová metoda pro transfer dat z InputStream do OutputStream:

```
var classLoader = ClassLoader.getSystemClassLoader();
var inputStream = classLoader.getResourceAsStream("myFile.txt");
var tempFile = File.createTempFile("myFileCopy", "txt");
try (var outputStream = new FileOutputStream(tempFile)) {
   inputStream.transferTo(outputStream);
}
```

Drobnosti – práce s diskem:

- ByteArrayOutputStream.writeBytes(byte[]);
- Files.readString(Path);
- Files.writeString(Path, String);
- Path.of(String)

Drobnosti: Predicate.not()

- Tohle:

```
lines.stream()
     .filter(s -> !s.isBlank())
```

- lze nyní přepsat takto:

```
lines.stream()
    .filter(Predicate.not(String::isBlank))
```

ZGC & Epsilon

- ZGC je Garbage Collector, který má za cíl GC pauzy pod 10 ms a škálovatelnost (zvládá efektivně fungovat i na X TB heap):
  - https://openjdk.java.net/jeps/333
  - https://wiki.openjdk.java.net/display/zgc/Main
  - Od Java 15 je production ready
- Epsilon (No-Op GC) alokuje paměť, ale neprovádí žádnou dealokaci paměti. Může být vhodný u aplikací, u kterých víme kolik paměti se bude maximálně alokovat a nechceme aplikaci zpomalovat zbytečným během GC.
  - https://openjdk.java.net/jeps/318

Další

- https://advancedweb.hu/2019/02/19/post_java_8/
- https://www.azul.com/90-new-features-and-apis-in-jdk-11/