

# An introduction to Java 9

*How can your applications benefit  
from Java 9?*



@IoannisKolaxis

Senior Expert / Software Engineer @ Unify

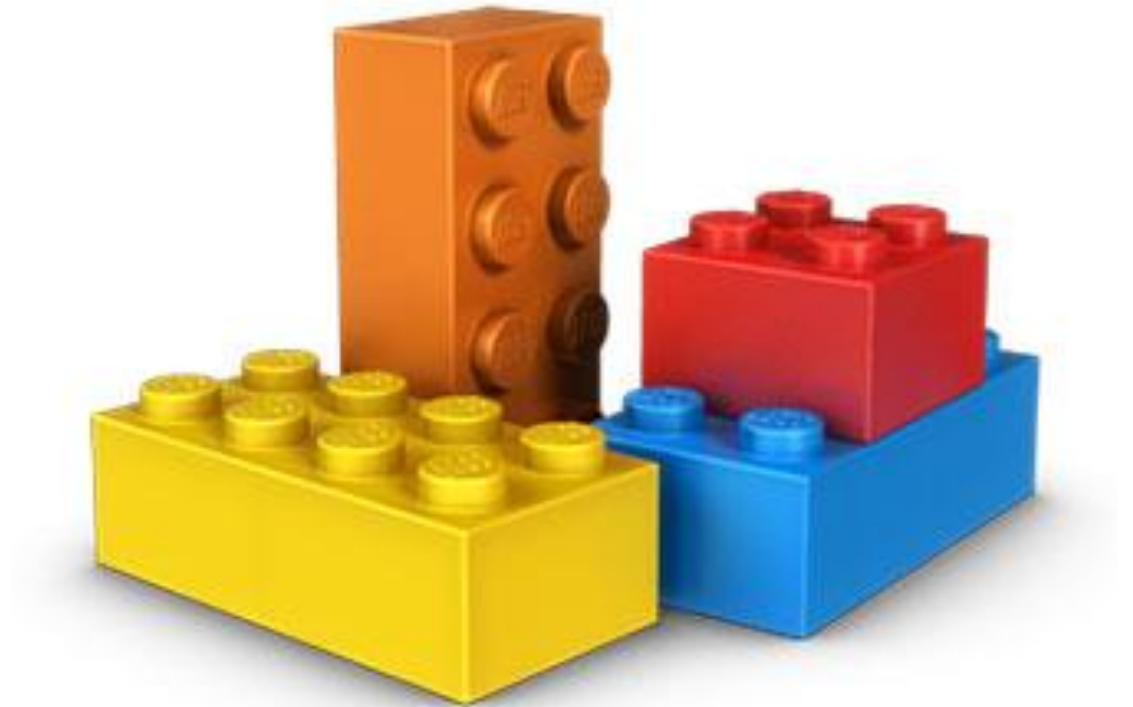
Java Meetup – Thessaloniki, 7<sup>th</sup> Nov 2017



# Java 9

The most important feature:

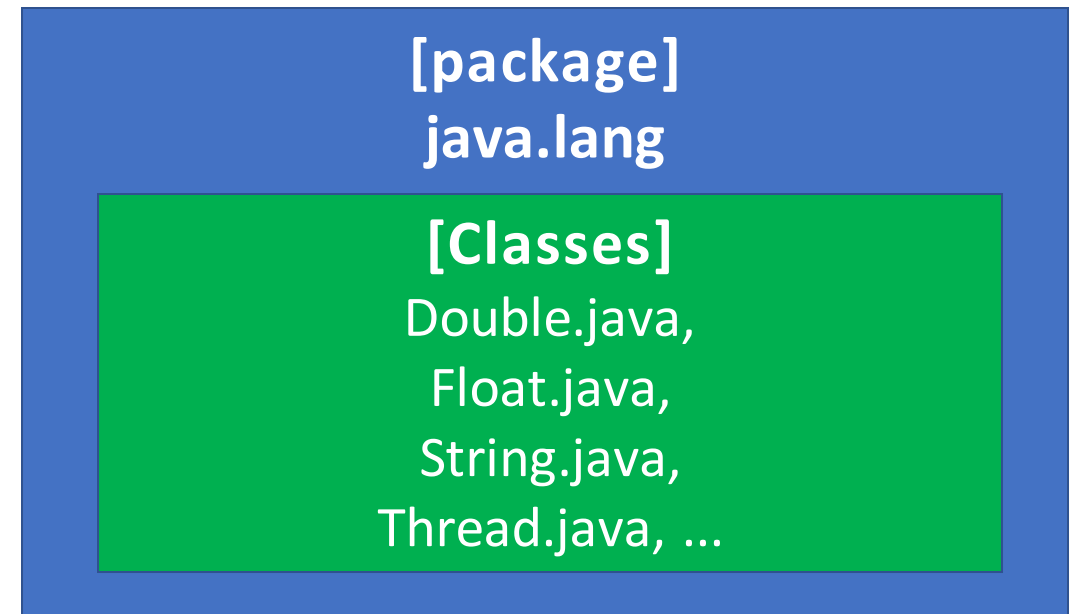
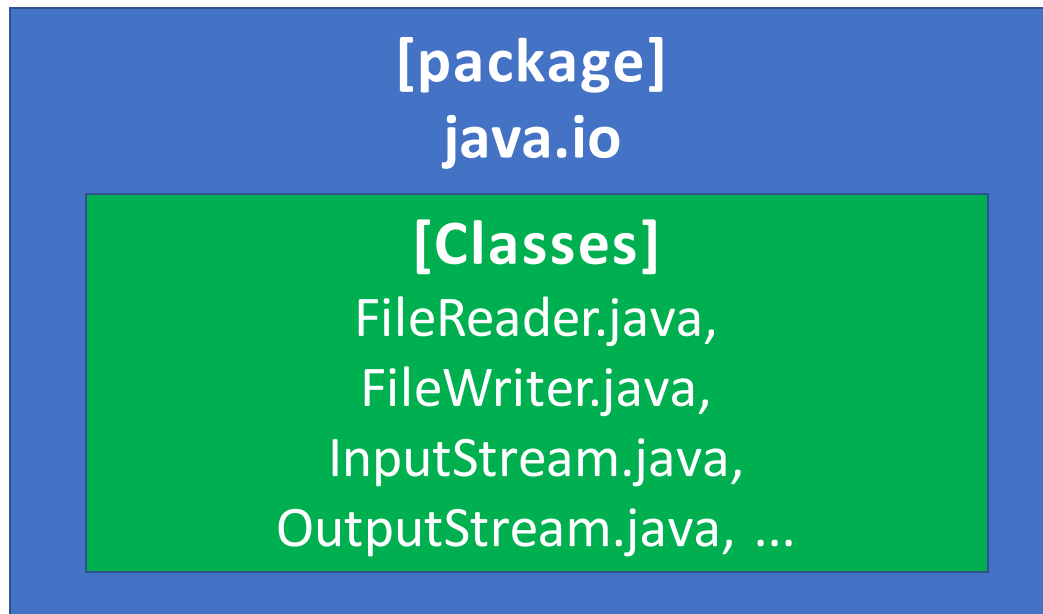
# Modules!



# What is a **package**?

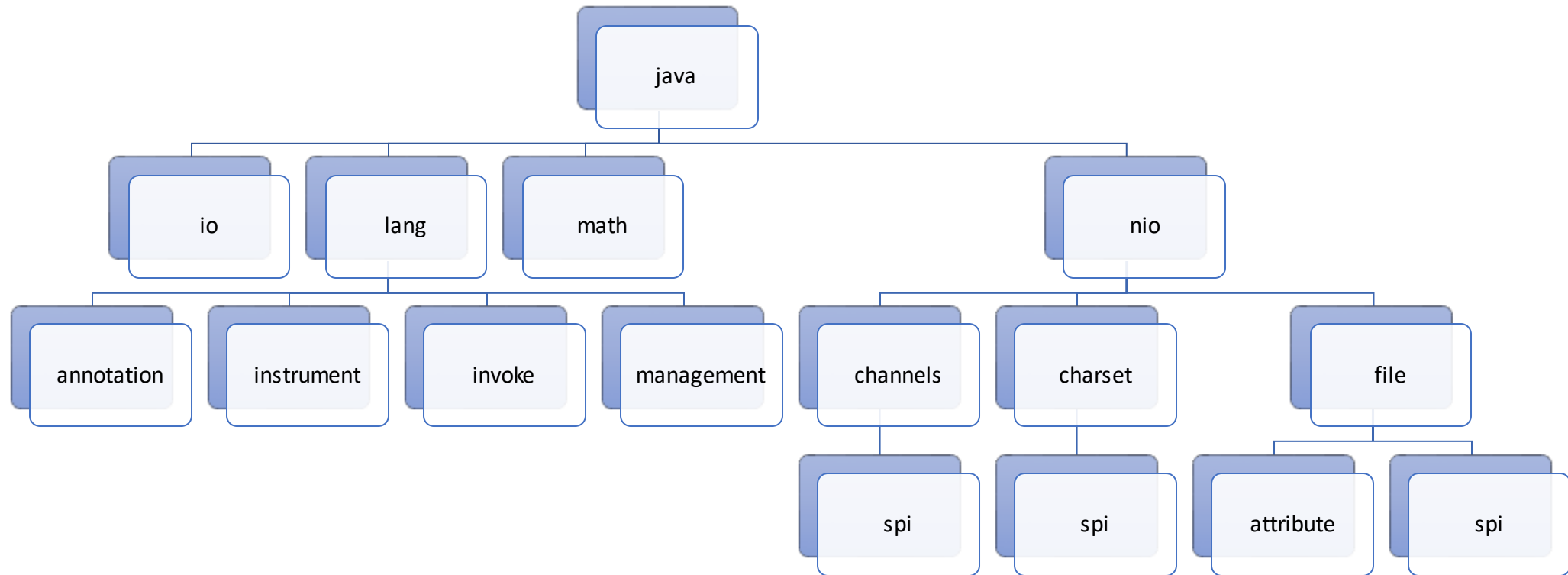
A package is a set of classes

- **Packages** are used to organize **classes & interfaces**



Quiz: How many **packages** does JDK 8 have?

A. 67      B. 117      C. 217

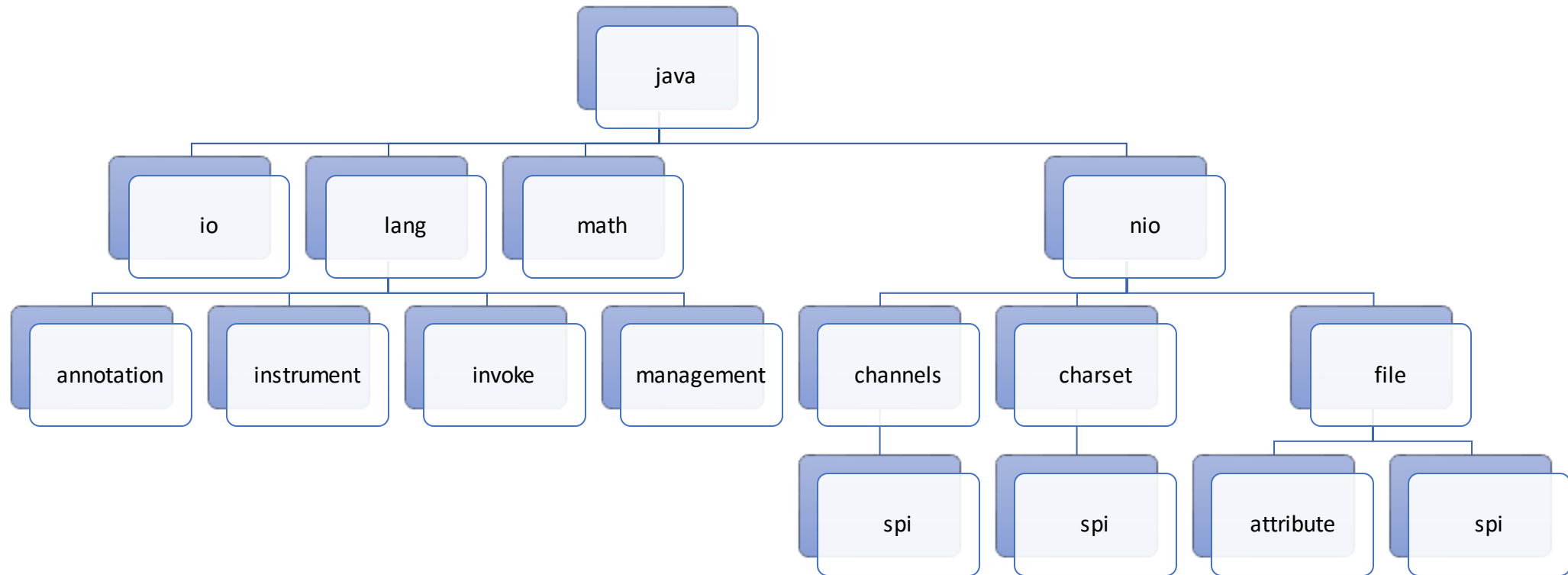


Quiz: How many **packages** does JDK 8 have?

A. 67

B. 117

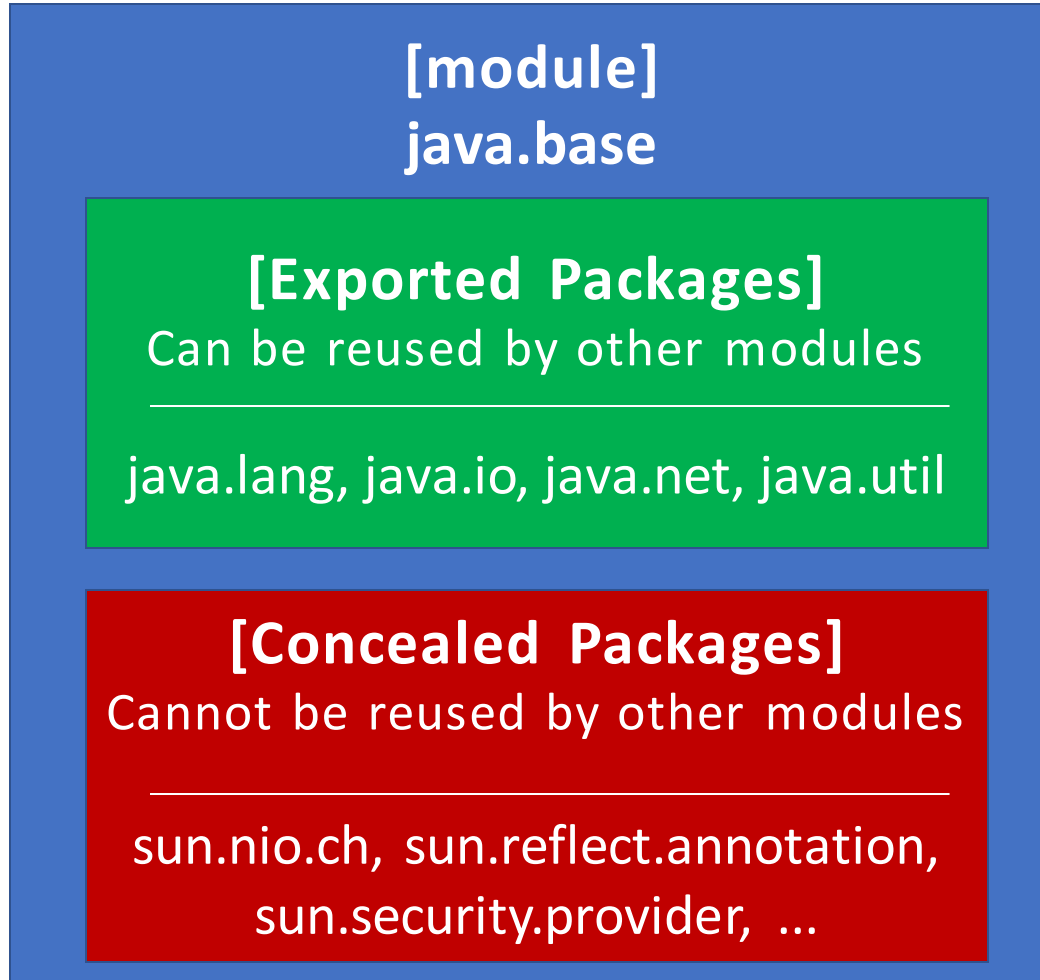
C. 217



# Challenges faced by Java engineers

- How do the classes of those 217 packages *interact* between them?
  - A **public class** is *accessible* by every other class, in any other package.
  - *Tight coupling* makes it too difficult to introduce changes, resulting in increased development & testing costs.
- Can we create *groups of packages*, controlling how they *interact* between them?

# Java 9: What is a **module**?

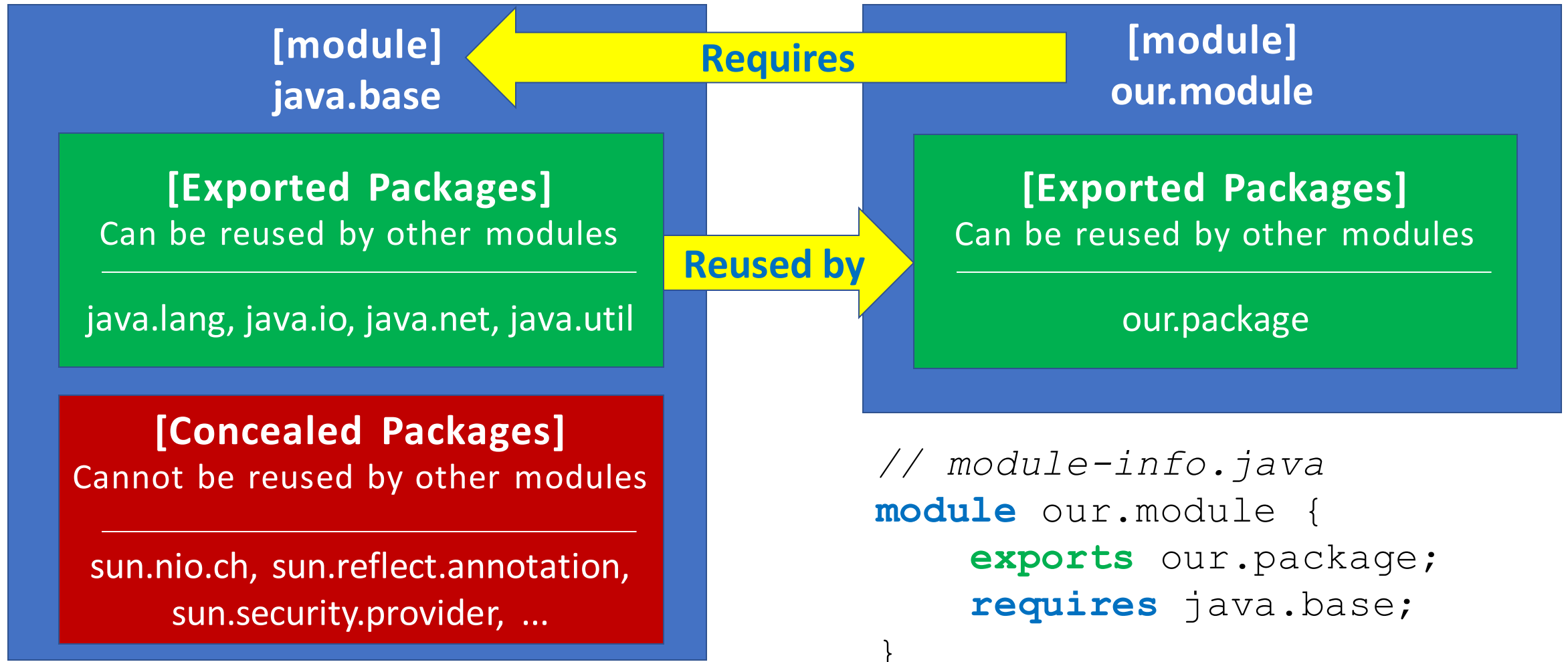


A module is a  
set of packages

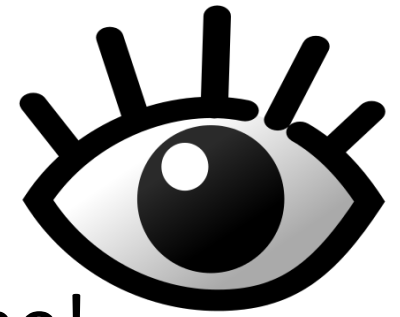
... designed  
for reuse

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

# Reusing packages from another module







# Using modules to control access

- In Java 9, a **public class** may not be *visible* to everyone!
- A **public class** in *our.package* may be:

```
module our.module {  
    exports our.package;  
}
```

Accessible to everyone

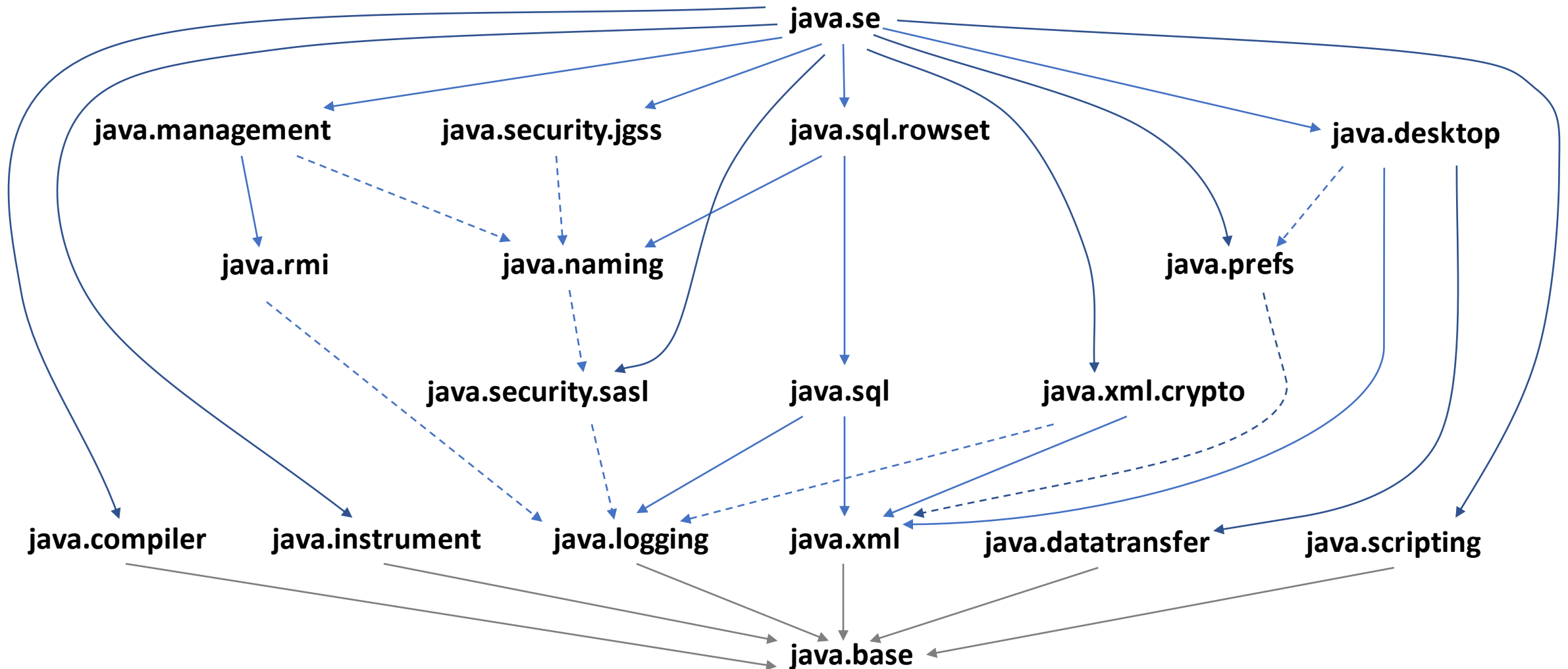
```
module our.module {  
    exports our.package  
        to friend.module;  
}
```

Accessible to other classes in our module & a friend module

```
module our.module {  
}
```

Accessible only to other classes in our module

# Modularizing Java 9



# Tailoring the JRE to fit **our** needs

- Are you worried about the **size** of the **JRE**?
- Are you building:
  - **Embedded**, or
  - **Containerized** applications?
- If yes, then *you* can **minimize** *your* JRE:
  - By deploying **only** the JRE modules that **you** need!

# Using the **Java Linker** to minimize our JRE

- Let's suppose that our application needs only the **java.base** module:

```
jlink --module-path /opt/jdk-9/jmods  
      --add-modules java-base  
      --output /out/jdk-9-base
```



- As demonstrated [here](#), the size of a Docker image of Alpine Linux & JRE 9 is reduced:
  - From: **356MB** (JRE with all modules)
  - To: **37MB** (JRE with **java.base** module only)

# How can your apps benefit from modules?

- Enhanced **security**, through *strong encapsulation*.
- **Performance** is improved:
  - only the required JRE modules may be loaded.
  - the classloader does not have to linearly search through all the JARs in the classpath, in order to find a class.

# Creating collections made easy in Java 9!

- How would you create a small, unmodifiable collection?

```
Set<String> set = new HashSet<>();  
set.add("a");  
set.add("b");  
set.add("c");  
set = Collections.unmodifiableSet(set);
```

≤ Java 8

- *List*, *Map*, and *Set* interfaces are enhanced in Java 9 with static factory methods “*of*”:

```
Set<String> set = Set.of("a", "b", "c");
```

Java 9

# Creating collections made easy in Java 9!

- More examples:

```
List<String> list = List.of("m", "e", "s");  
Map<String, Integer> map = Map.of("e", 5, "s", 1);
```

Collections are  
immutable!

- What about *Maps* with more than 10 elements?

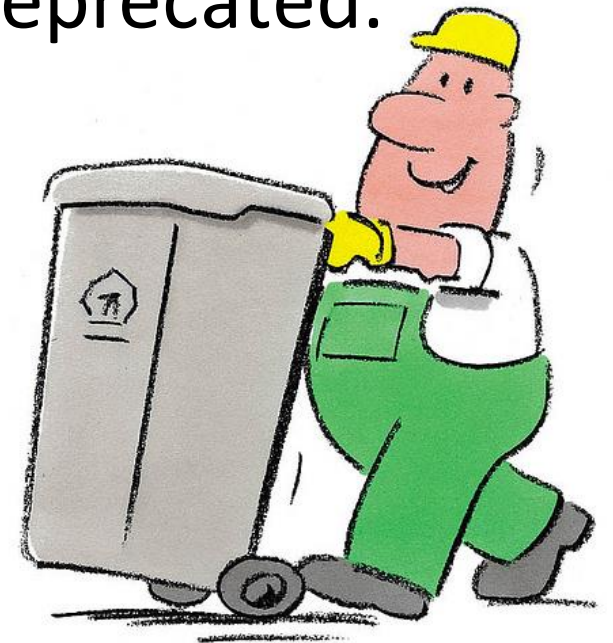
```
Map<String, Integer> map = Map.ofEntries(  
    entry("a", 1),  
    entry("b", 2),  
    ...  
    entry("z", 26));
```

... cannot  
add/delete  
elements

- For *Sets*, and *Maps* the iteration order is randomized!

# Java 9: Changes to Garbage Collectors (GC)

- ***Garbage-First (G1)*** becomes the **default** Garbage Collector.
  - Previously, the default one was the **Parallel** GC.
- *Concurrent Mark Sweep (CMS)* GC becomes deprecated.
- How will those changes affect your existing application?





# What does your application aim for?

- Low Latency: Responding *quickly*, in *short periods of time*.
  - e.g. when serving a web page, or returning a database query.

*or*
- High Throughput: Maximizing the *amount of work* done in a given time frame.
  - e.g. number of database queries completed in 1 hour.



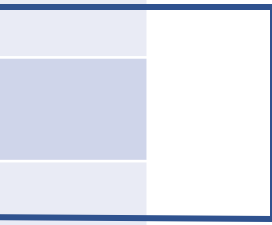
# Tuning the performance of your application

- As Java programmers, we don't need to manage memory:
  - We *allocate* memory for an object with *new*:  
`String meetup = new String("jhug");`
  - This memory will be *automatically deallocated* by the Garbage Collector, when it is no longer in use.
- How do we tune the *performance* of an application?
  - Heap: by properly sizing the heap (the memory area where all the object data is stored).
  - Garbage Collector: by choosing the most appropriate GC, usually optimized for *Low Latency* or *High Throughput*.

# Moving to a Low-Latency Garbage Collector

- Garbage Collectors in Java 9:

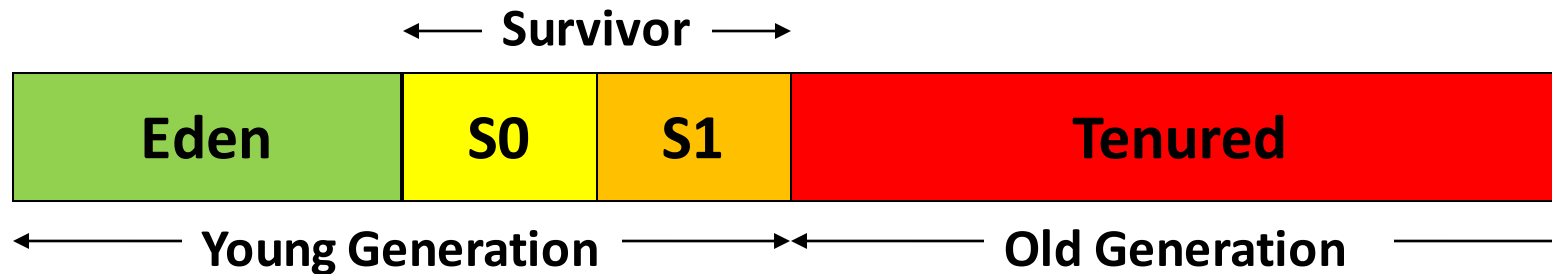
Java 9	Garbage Collector	Optimized for
	Serial	Memory footprint
(Ex-Default)	Parallel	High Throughput
Deprecated	Concurrent Mark Sweep	Low Latency
Default	Garbage First / G1	Low Latency



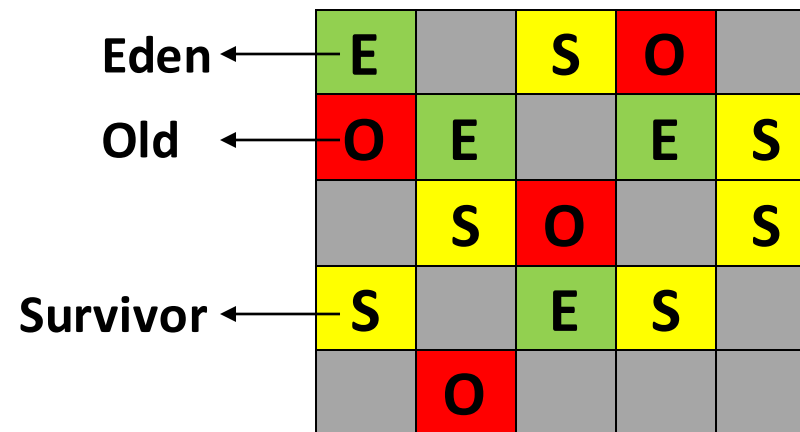
- Moving from a *Throughput-oriented* GC (Parallel) to a *Low-Latency* GC (Garbage First/G1).

# G1 Garbage Collector

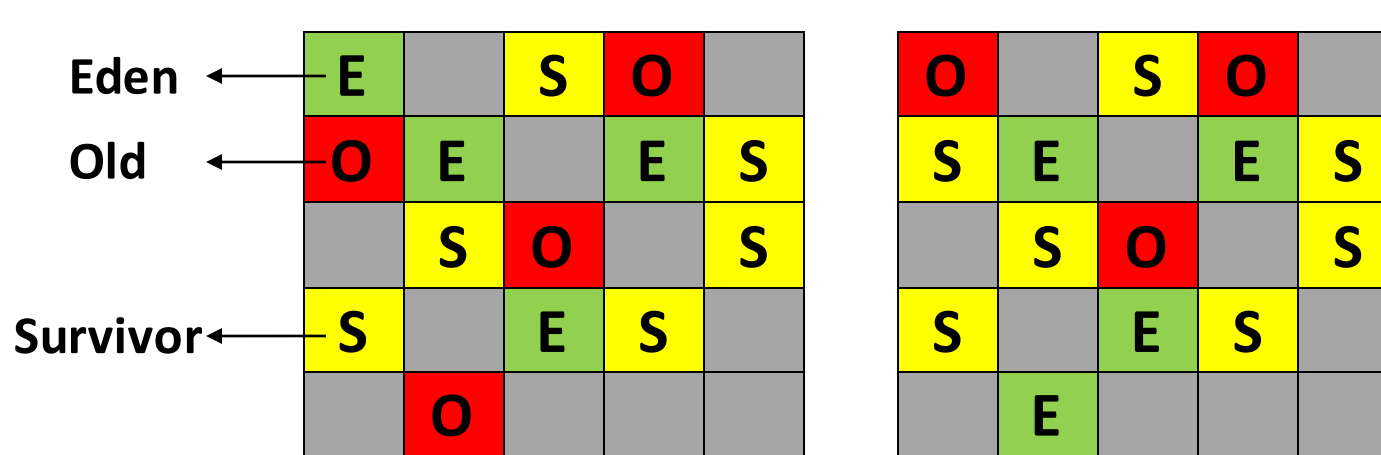
- In Serial/Parallel/CMS GC, the Heap is divided into **2 regions** (=Young & Old Generation) of **fixed** size.



- In G1 GC, the Heap is divided into **multiple, smaller regions**.



# G1 Garbage Collector

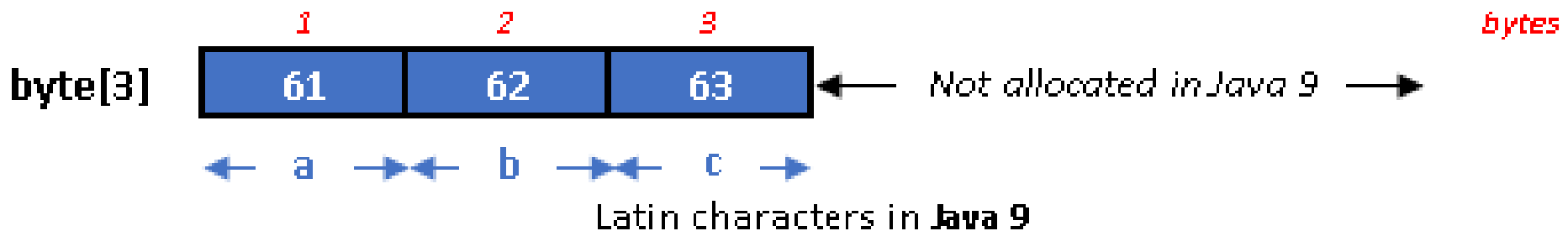
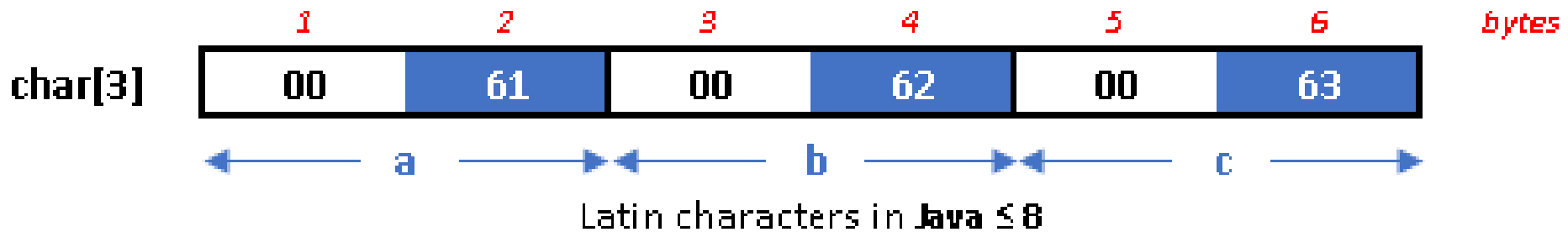
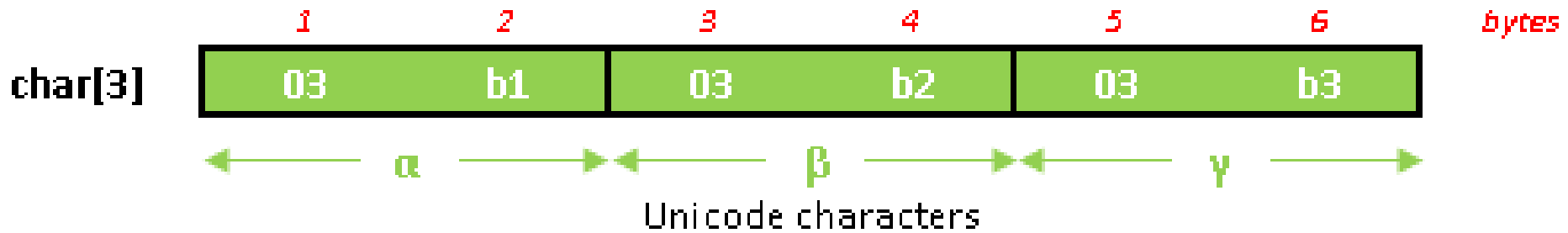


Region size  
depends on  
Heap Size

6GB heap : 2.000 regions  
= **3MB/region**

- The heap is partitioned into *2.000 equal-sized **regions***:
  - Achieving a finer granularity on how much garbage to collect at each GC pause.
- Default pause goal = *200msec*
  - May be adjusted, to achieve lower latency or higher throughput.

# Compact Strings (JEP 254)



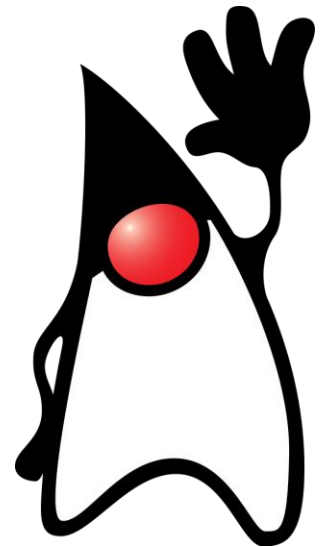
# Compact Strings - Will your apps run *faster*?

- *Allocation rate*: the amount of memory allocated per time unit, measured in MB/sec.
- Strings now have a *reduced memory footprint*, resulting in:
  - lower memory allocation rate,
  - the GC is called less frequently to clean the memory,
  - a decrease in the frequency and/or duration of pauses during GC cleanup.
- Performance improvements of *up to 10%* have been measured, as stated [here](#).



# Controlling native processes

- Ever had to control native processes from your Java app?
  - Executed OS commands (like: `'ps -ef'`), and parsed their output?
- Java 9 makes it a “piece of cake” to:
  - Get the native id of the current process:  
`long processId = ProcessHandle.current().pid();`
  - Check if a process is currently running.
  - Retrieve information (like: start/total time) for a process.
  - Wait for termination of a process, and trigger dependent actions :  
`process.onExit().thenRun(() -> System.out.println("Done"));`





# HTTP/2 Client API

- Until Java 8, we used the `HttpURLConnection` API to establish an HTTP 1.1 connection. However:
  - It works only in blocking mode (=1 thread per request/response).
  - It is hard to use, with many undocumented behaviors.
- **Java 9** introduces a new HTTP client that supports:
  - HTTP 1.1 and **HTTP/2**,
  - a synchronous/blocking mode, and an asynchronous/non-blocking mode.
- Delivered as an *incubator* module
  - A non-final API, provided for experimentation.
  - May be finalized (or even removed!) in an upcoming Java version.

# Security related enhancements

- Improved **performance** for **GHASH** and **RSA** cryptographic operations (JEP 246).
  - By leveraging SPARC and Intel x64 CPU instructions.
- Support for SHA-3 hash algorithms (JEP 287).
- TLS Application-Layer Protocol Negotiation (JEP244)
  - Provides the means to negotiate an application protocol over TLS.
  - Required by HTTP/2.
- OCSP Stapling for TLS (JEP 249).
  - Improves performance of TLS, by reducing the performance bottleneck of the OCSP responder.



# Java Shell: Read-Evaluate-Print Loop (REPL)

- REPL - an interactive programming tool, that:
  - Loops, continuously reading user input,
  - Evaluates the input,
  - Prints the value of the input.
- Expected to help students learn Java, without having to *edit*, *compile*, and *execute* code.
- As developers, we can benefit from JShell by quickly:
  - Exploring new APIs or language features,
  - Prototyping something complex.

# Wish to learn more about Java 9?

- For more details, check the **JEPs** (Java Enhancement Proposals) implemented in Java 9:

<http://openjdk.java.net/projects/jdk9/>

## OpenJDK

OpenJDK FAQ  
Installing  
Contributing  
Sponsoring  
Developers' Guide

Mailing lists  
IRC · Wiki  
Bylaws · Census  
Legal

### JEP Process

### Source code

Mercurial  
Bundles (6)

### Groups

(overview)  
2D Graphics  
Adoption  
AWT  
Build

## JDK 9

The goal of this Project was to produce an open-source reference implementation of the Java SE 9 Platform as defined by JSR 379 in the Java Community Process.

JDK 9 reached General Availability on 21 September 2017. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal.

### Features

102: Process API Updates  
110: HTTP 2 Client  
143: Improve Contended Locking  
158: Unified JVM Logging  
165: Compiler Control

# References

1. [JDK 9 Features](#)
2. [“Modular Development with JDK 9”, Alex Buckley, Devovx United States](#)
3. [“Java in a World of Containers”, Paul Sandoz](#)
4. [“The G1 GC in JDK 9”, Erik Duveblad](#)
5. [“JShell is Here”, Robert Field](#)