

# Application Delivery Fundamentals : Java

Java 9 to 21



High performance. Delivered.

# Java 8

## Major Features

### Lambdas

```
s -> do(s)  
Eg.  
arr.forEach((String s) ->  
    System.out.print(s));  
arr.forEach(s->  
    System.out.print(s));  
arr.forEach(  
    System.out::print);
```



### Java 8 (LTS)

### Java 9

### Modularity

```
module my.module {  
    requires module.name;  
    exports my.moduel.api;  
}
```

### Unified logging

```
private static System.Logger LOGGER =  
System.getLogger("MyApp");  
  
LOGGER.log(Level.ERROR, "error test");
```



### Java 10

### Var improved inferencing

```
var a = new ArrayList<String>()  
var stream = list.stream();
```



### Java 11 (LTS)

### Var on Lambdas

```
var s -> do(s)
```

## APIs Features

### Lambdas compatibility

```
arrays.forEach(s -> print(s))  
Consumer, Function, Supplier,  
Predicate, Operator interfaces
```

### Stream, Parallel Stream

```
myList .stream() .filter(s ->  
s.startsWith("B"))  
.map(String::toLowerCase)  
.sorted()  
.forEach(System.out::println)
```

### String

```
String.join("B",list);
```

### Optional

```
Optional<String> optional =  
Optional.of(str);
```

### Immutable List

```
List.of()  
Set.of()
```

### Stream, Parallel Stream

```
takeWhile() and dropWhile()
```

### String

```
Implementation as byte[]
```

### Optional

```
Optional.isPresentOrElse()
```

### Lambdas compatibility

```
arrays.forEach(s -> print(s))
```

### Immutable List

```
List.copyOf()  
Set.copyOf()
```

### Optional

```
Optional.orElseThrow()
```

### String

```
String.strip()  
String.isBlank()  
String.readString(path file)
```

### Stream, Parallel Stream

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

### Optional

```
Optional.of(obj).isEmpty()
```

## JVM/Lang.

### Interface: default method and static

### JShell: command line interpreter

### Interface: private methods

### GC: G1 is the default GC

### Module: java --describe-module \$module

```
Logging: unified logging with -Xlog  
java -Xlog:all=debug:file=application.log -  
version
```

### JIT: new experimental

### javah removed

```
Improvements of repository and thread-local
```

### GC

- Epsilon GC: the GC no-opt
- JAVA EE: deprecated

```
java HelloWorld.java (without compilation) and shebang
```

### Interface: nested class

# Goals

---

- Java Versioning
- Java 9 changes
- Collection and stream updates
- Java 9 concurrency updates
- Other new java features
- The java module system (jigsaw)
- Introduction to the module system
- The module descriptor

# Goals

---

- Working with modules
- Jshell
- Java 10 changes
- Local-variable type inference
- Java 11 changes
- Using strings in java 11
- Java 11: removed features and options
- The http client api

## Goals

---

- Additional topics memory management
- Performance optimizations

# Versioning

---

Version	Release Data	Notes
JDK 1.0	January 1996	
J2SE 5.0	September 2004	5 Releases in 8 years
Java SE 8 (LTS)	March 2014	Most important Java Release
Java SE 9	September 2017	4 Releases in 13 years
Java SE 10	March 2018	Time-Based Release Versioning
Java SE 11 (LTS)	September 2018	Long Term Support Version (Every 3 years)
Java SE 12	March 2019	
...		
Java SE 16	March 2021	

# Java New Features

---

Version	Release Date	Important New Features
J2SE 5.0	Sep 2004	Enhanced For Loop, Generics, Enums, Autoboxing
Java SE 8 (LTS)	Mar 2014	Functional Programming - Lambdas & Streams, Static methods in interface
Java SE 9	Sep 2017	Modularization (Java Platform Module System)
Java SE 10	Mar 2018	Local Variable Type Inference
Java SE 14	Mar 2020	Switch Expressions (Preview in 12 and 13)
Java SE 15	Sep 2020	Text Blocks (Preview in 13)
Java SE 16	Mar 2021	Record Classes (Preview in 14 and 15)
All Java Versions	-	API Improvements, Performance and Garbage Collection Improvements

## Java 9 Underscore

---

- In Java 9 release, underscore is a keyword and can't be used as an identifier or variable name.
- If we use the underscore character ("\_") as an identifier, our source code can no longer be compiled.

## Java 9 Anonymous class Inference

---

- Java 9 introduced a new feature that allows us to use diamond operator with anonymous classes.
- Using the diamond with anonymous classes was not allowed in Java 7.
- In Java 9, as long as the inferred type is denotable, we can use the diamond operator when we create an anonymous inner class.
- Data types that can be written in Java program like int, String etc are called denotable types. Java 9 compiler is enough smart and now can infer type.

## @SafeVarargs Annotation

---

- It is an annotation which applies on a method or constructor that takes varargs parameters.
- It is used to ensure that the method does not perform unsafe operations on its varargs parameters.
- It was included in Java7 and can only be applied on
  - Final methods
  - Static methods
  - Constructors
- From Java 9, it can also be used with private instance methods.

## Private methods in interfaces

---

- In Java 9, we can create private methods inside an interface.
- Interface allows us to declare private methods that help to share common code between non-abstract methods.

# JAVA 9 CHANGES

---

- `@Deprecated`
  - Optional Attributes Added in Java 9
- Java 9 adds some optional attributes to the `@Deprecated` annotation: `since` and `forRemoval`.
- The `since` attribute requires a string that lets us define in which version the element was deprecated. The default value is an empty string.
- And `forRemoval` is a boolean that allows us to specify if the element will be removed in the next release. Its default value is false:

# Java Deprecated API Scanner tool (jdepscan) in Java 9

---

- Java Deprecated API Scanner tool i.e. jdeprscan is a static analyzing command-line tool which is introduced in JDK 9 for find out the uses of deprecated API in the given input.
- Here the input can be .class file name, directory or JAR file.
- Whenever we provide any input to jdeprscan command line tool then it generates the dependencies to the system console.
- jdeprscan introduced with various options which affect the output.
- As per option, jdeprscan command-line tool generates the output.
- jdeprscan tool identifies the deprecated APIs which defined by Java SE Deprecated APIs but it will not list out the deprecated APIs which is used by third-party libraries.

# Java Deprecated API Scanner tool (jdeprscan) in Java 9

```
C:\Windows\System32\cmd.exe
com\boa\simpleexercises\SafeArgsTest.class

C:\Program Files\Java\jdk-11.0.12\bin>jdeprscan --verbose F:\java9ws\simpleexercises\targe
t\classes\com\boa\simpleexercises\SafeArgsTest.class
List of classes to process:
java.awt.image.AbstractMultiResolutionImage
java.awt.image.AffineTransformOp
java.awt.image.AreaAveragingScaleFilter
java.awt.image.BandCombineOp
java.awt.image.BandedSampleModel
java.awt.image.BaseMultiResolutionImage
java.awt.image.BufferStrategy
java.awt.image	BufferedImage
java.awt.image.BufferedImageFilter
java.awt.image.BufferedImageOp
java.awt.image.ByteLookupTable
java.awt.image.ColorConvertOp
java.awt.image.ColorModel
java.awt.image.ComponentColorModel
java.awt.image.ComponentSampleModel
java.awt.image.ConvolveOp
java.awt.image.CropImageFilter
```

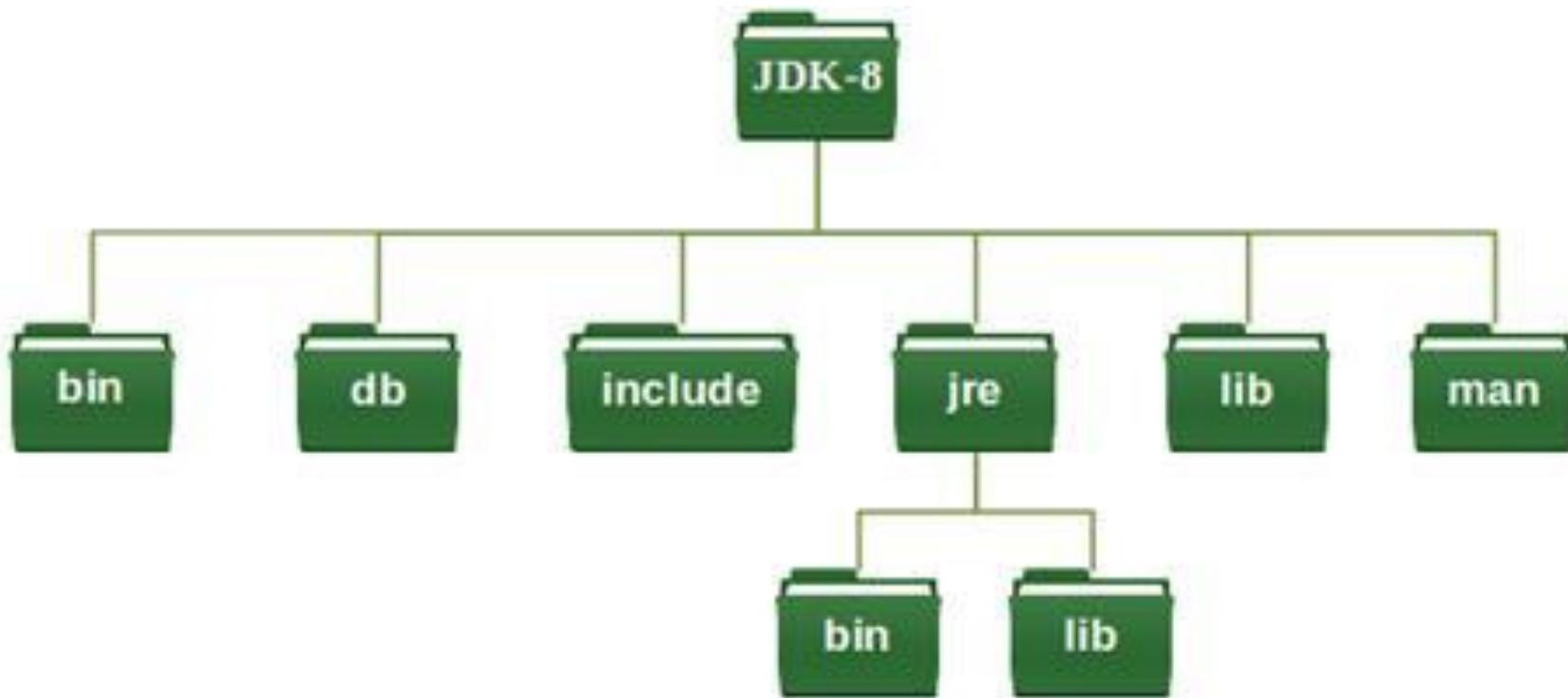


# Java Deprecated API Scanner tool (jdeprscan) in Java 9

```
C:\Windows\System32\cmd.exe
C:\Program Files\Java\jdk-11.0.12\bin>jdeprscan --verbose F:\java9ws\simpleexercises\target\classes\com\boa\simpleexercises\SafeArgsTest.class
List of classes to process:
java.awt.image.AbstractMultiResolutionImage
java.awt.image.AffineTransformOp
java.awt.image.AreaAveragingScaleFilter
java.awt.image.BandCombineOp
java.awt.image.BandedSampleModel
java.awt.image.BaseMultiResolutionImage
java.awt.image.BufferStrategy
java.awt.image.BufferedImage
java.awt.image.BufferedImageFilter
java.awt.image.BufferedImageOp
java.awt.image.ByteLookupTable
java.awt.image.ColorConvertOp
java.awt.image.ColorModel
java.awt.image.ComponentColorModel
java.awt.image.ComponentSampleModel
java.awt.image.ConvolveOp
java.awt.image.CropImageFilter
java.awt.image.DataBuffer
java.awt.image.DataBufferByte
```

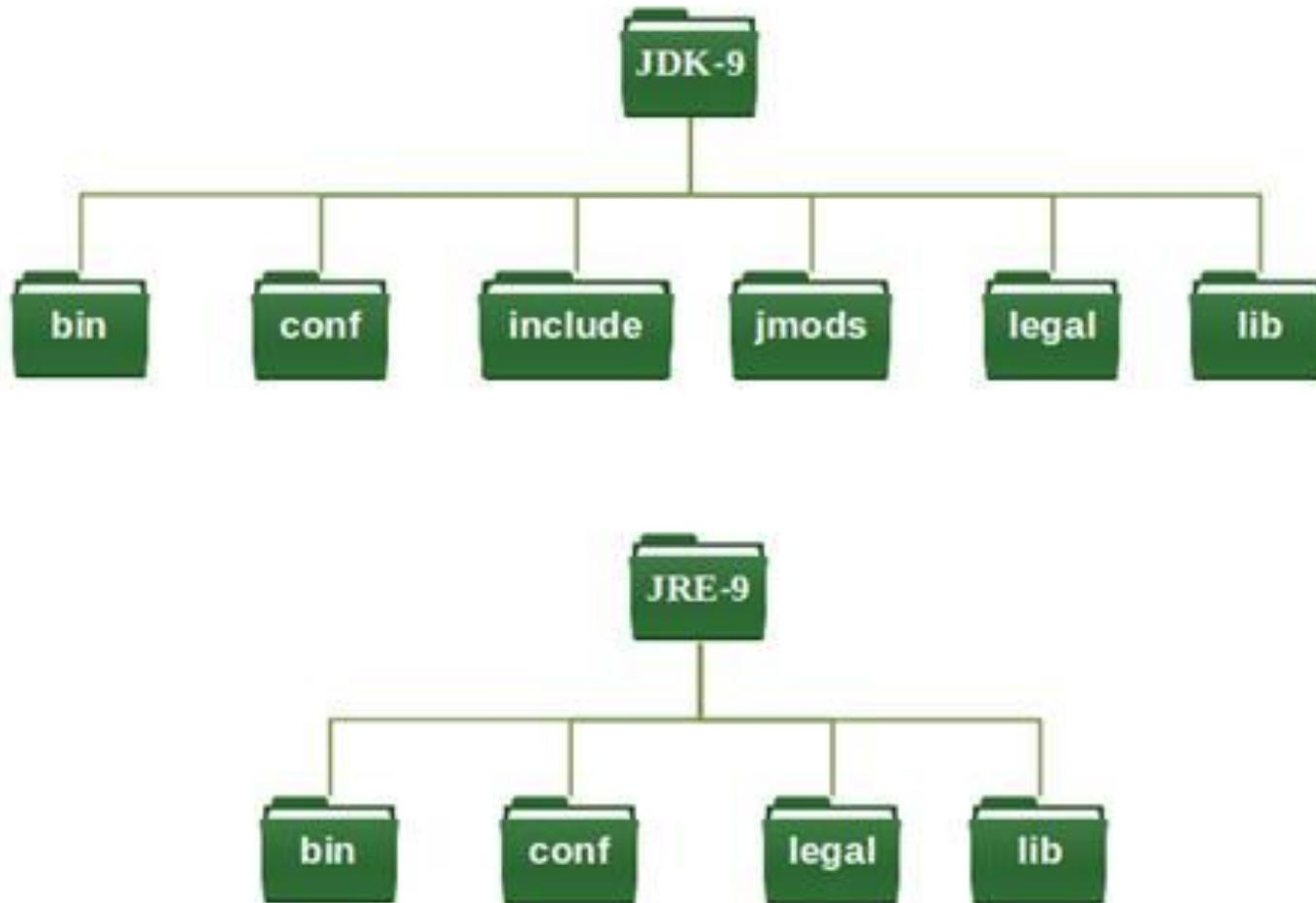
# Java 9 JDK/JRE Structural Changes

---



# Java 9 JDK/JRE Structural Changes

---



## JDK Modules

---

- The entire JDK has been divided into a set of modules.
- As a result, one can pick a set according to the requirement either during compilation, build time, or during run time into a variety of configurations.
- The module name specific to JDK begins with “jdk”.  
The list with descriptions is as follows (\*).

## JMODs

---

- JMOD enables aggregating files other than class files, metadata, and resources such as native codes and other things that cannot be stored in a JAR file.
- Therefore, JMOD files are designed to contain file types that cannot be contained by JAR files.
- JARs are executable, but the JMOD files cannot be executed.
- This means this files contained in JMOD can be used only at compile-time or link-time, but not at runtime.
- We use the jmod tool to create JMOD files and list the content of existing JMOD files.

# JDK Modules

---

Module	Description
jdk.accessibility	Defines JDK utility classes used by implementers of Assistive Technologies.
jdk.attach	Defines the attach API.
jdk.charsets	Provides charsets that are not in java.base (mostly double byte and IBM charsets).
jdk.compiler	Defines the implementation of the system Java compiler and its command line equivalent, <i>javac</i> , as well as <i>javah</i> .
jdk.crypto.cryptoki	Provides the implementation of the SunPKCS11 security provider.
jdk.crypto.ec	Provides the implementation of the SunEC security provider.
jdk.dynalink	Defines the API for dynamic linking of high-level operations on objects.
jdk.editpad	Provides the implementation of the edit pad service used by <i>jdk.jshell</i> .
jdk.hotspot.agent	Defines the implementation of the HotSpot Serviceability Agent.
jdk.httpserver	Defines the JDK-specific HTTP server API.
jdk.incubator.httpclient	Defines the high-level HTTP and WebSocket API.
jdk.jartool	Defines tools for manipulating Java Archive (JAR) files, including the <i>jar</i> and <i>jarsigner</i> tools.
jdk.javadoc	Defines the implementation of the system documentation tool and its command line equivalent, <i>javadoc</i> .
jdk.jcmd	Defines tools for diagnostics and troubleshooting a JVM, such as the <i>jcmd</i> , <i>jps</i> , and <i>jstat</i> tools.

# JDK Modules

jdk.jconsole	Defines the JMX graphical tool, <i>jconsole</i> , for monitoring and managing a running application.
jdk.jdeps	Defines tools for analysing dependencies in Java libraries and programs, including the <i>jdeps</i> , <i>javap</i> , and <i>jdeprscan</i> tools.
jdk.jdi	Defines the Java Debug Interface.
jdk.jdwp.agent	Provides the implementation of the Java Debug Wire Protocol (JDWP) agent.
jdk.jlink	Defines the <i>jlink</i> tool for creating run-time images, the <i>jmod</i> tool for creating and manipulating JMOD files, and the <i>jimage</i> tool for inspecting the JDK implementation-specific container file for classes and resources.
jdk.jshell	This module provides support for Java Programming Language ‘snippet’ evaluating tools, such as Read-Eval-Print Loops (REPLs), including the <i>jshell</i> tool.
jdk.jsobject	Defines the API for the JavaScript Object.
jdk.jstatd	Defines the <i>jstatd</i> tool for starting a daemon for the <i>jstat</i> tool to monitor JVM statistics remotely.
jdk.localedata	Provides the locale data for locales other than the US.
jdk.management	Defines JDK-specific management interfaces for the JVM.
jdk.management.agent	Defines the JMX management agent.
jdk.naming.dns	Provides the implementation of the DNS Java Naming provider.
jdk.naming.rmi	Provides the implementation of the RMI Java Naming provider.
jdk.net	Defines the JDK-specific Networking API.
jdk.pack	Defines tools for transforming a JAR file into a compressed pack200 file and transforming a packed file into a JAR file, including the <i>pack200</i> and <i>unpack200</i> tools.

# JDK Modules

jdk.packager.services	Defines the services used by the Java packager tool.
jdk.policytool	Defines the GUI tool for managing policy files, called <i>policytool</i> .
jdk.rmic	Defines the <i>rmic</i> compiler for generating stubs and skeletons using the Java Remote Method Protocol (JRMP) and stubs and tie class files (IIOP protocol) for remote objects.
jdk.scripting.nashorn	Provides the implementation of Nashorn script engine and the runtime environment for programs written in ECMAScript 5.1.
jdk.sctp	Defines the JDK-specific API for SCTP.
jdk.security.auth	Provides implementations of the javax.security.auth.* interfaces and various authentication modules.
jdk.security.jgss	Defines Java extensions to the GSS-API and an implementation of the SASL GSSAPI mechanism.
jdk.snmp	Defines the SNMP management agent.
jdk.xml.dom	Defines the subset of the W3C Document Object Model (DOM) API that is not part of the Java SE API.
jdk.zipfs	Provides the implementation of the zip file system provider.

# Effect of Modular JDK

---

- Modular JDK provides the nifty freedom to combine JDK modules into a variety of configurations, according to the need.
- The idea is somewhat an augmentation of the concept of compact profiles introduced with Java 8.
- Custom configuration is particularly advantageous not only in creating scalable Java application but also shipping it in a runtime environment as a scalable platform.
- The configuration can have a specific set of modules as opposed to the Java platform as a monolithic artifact.
- This would leverage the performance, maintainability, and security of a Java application.
- The modular structure also introduced a new URI scheme for naming modules, classes, and resources stored in a runtime image.
- This strengthens the principle of encapsulation to a whole new level. Most of the JDK's internal APIs are hidden except a few critical ones, until some supported versions are introduced in future versions.

## Project Jigsaw - Modularity

---



Java 9

# Java Platform Module System (Project Jigsaw)

---

- It is a new kind of Java programming component that can be used to collect Java code (classes and packages).
- The main goal of this project is to easily scale down application to small devices.
- In Java 9, JDK itself has divided into set of modules to make it more lightweight.
- It also allows us to develop modular applications.

# Java Platform Module System (Project Jigsaw)

---

- Java Module System is a major change in Java 9 version.
- Java added this feature to collect Java packages and code into a single unit called module.
- In earlier versions of Java, there was no concept of module to create modular Java applications, that why size of application increased and difficult to move around.
- Even JDK itself was too heavy in size, in Java 8, rt.jar file size is around 64MB.
- To deal with situation, Java 9 restructured JDK into set of modules so that we can use only required module for our project.

# Java Platform Module System (Project Jigsaw)

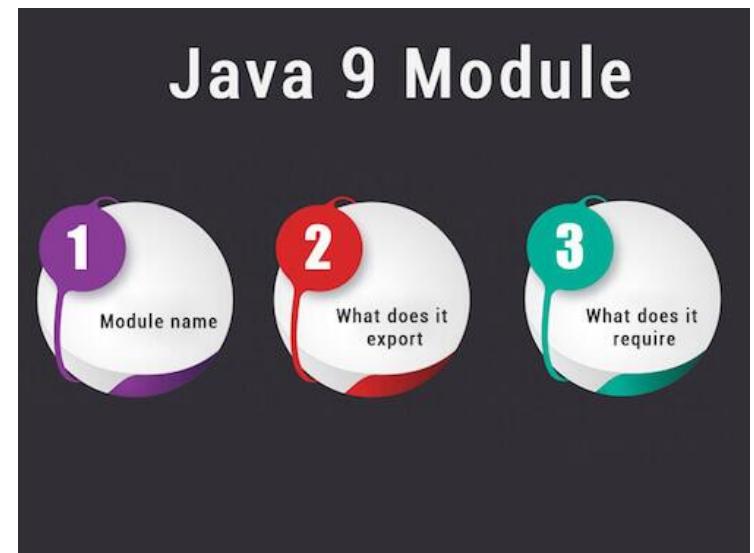
---

- The module system includes various tools and options that are given below.
- Includes various options to the Java tools javac, jlink and java where we can specify module paths that locates to the location of module.
- Modular JAR file is introduced. This JAR contains module-info.class file in its root folder.
- JMOD format is introduced, which is a packaging format similar to JAR except it can include native code and configuration files.
- The JDK and JRE both are reconstructed to accommodate modules. It improves performance, security and maintainability.
- Java defines a new URI scheme for naming modules, classes and resources.

# Java 9 Module

---

- Module is a collection of Java programs or softwares.
- To describe a module, a Java file `module-info.java` is required.
- This file also known as module descriptor and defines the following
  - Module name
  - What does it export
  - What does it require



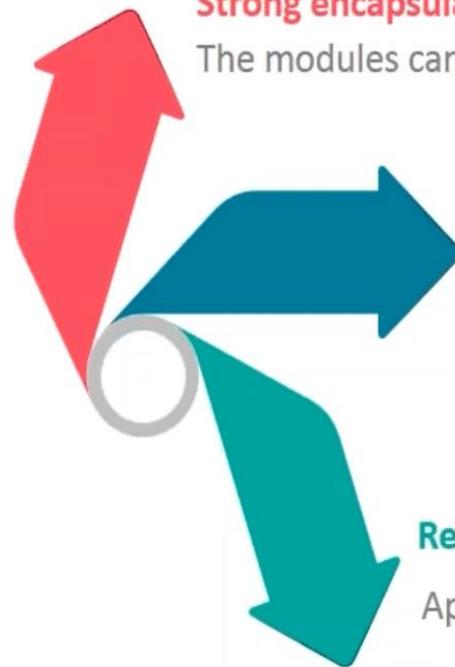
## Java 9 Module

---

- Module system is a part of Jigsaw Project. It adds one more abstraction level above packages.
- In other words, it is a ‘package of Packages’ that makes our code even more reusable.
- It is also fine to say that a module is a group of closely related packages, resources and module descriptor(module-info.java) file.
- In Java 9 ‘java.base’ is a base module. It does not depend on any other modules. By default, all modules including user defined modules are dependent on this module.

# Project Jigsaw – Benefits of Modularity

---



## Strong encapsulation

The modules can access only those parts of the module that have been made available for use

## Clear Dependencies

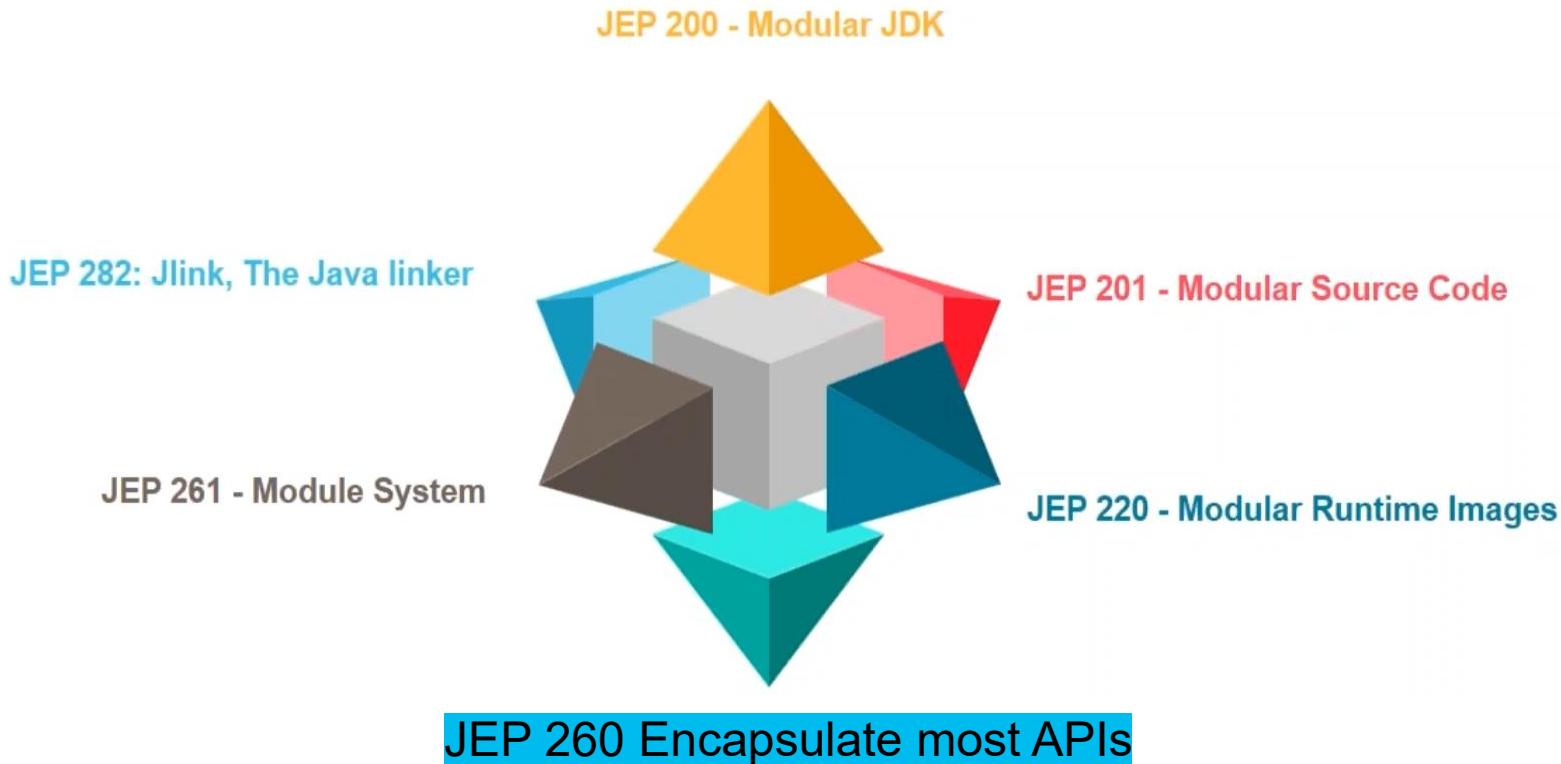
Provides the basis for a reliable configuration of modules

## Reliable

Applications are more reliable by eliminating run-time errors

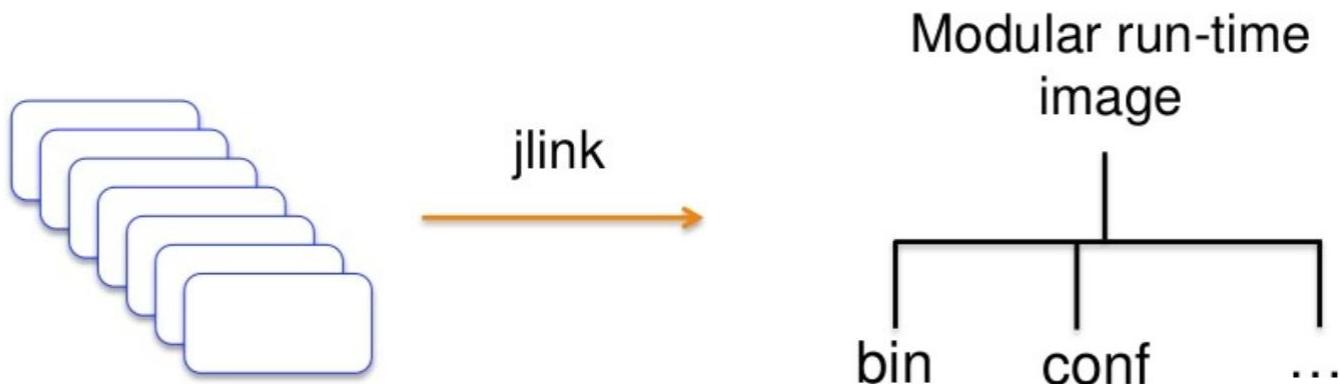
# Project Jigsaw – Benefits of Modularity

---



# Project Jigsaw – Benefits of Modularity

## jlink: The Java Linker (JEP 282)



```
$ jlink --modulepath $JDKMODS \  
--addmods java.base -output myimage
```

```
$ myimage/bin/java -listmods  
java.base@9.0
```

# Project Jigsaw – Benefits of Modularity

## jlink: The Java Linker (JEP 282)

```
$ jlink --modulepath $JDKMODS:$MYMODS \
--addmods com.azul.app -output myimage
```

```
$ myimage/bin/java -listmods
```

```
java.base@9.0
java.logging@9.0
java.sql@9.0
java.xml@9.0
com.azul.app@1.0
com.azul.zoop@1.0
com.azul.zeta@1.0
```

Version numbering for information purposes only  
“It is not a goal of the module system to solve the version-selection problem”

# Project Jigsaw – Benefits of Modularity

## jlink: The Java Linker (JEP 282)

```
$ jlink --modulepath $JDKMODS:$MYMODS \
--addmods com.azul.app -output myimage
```

```
$ myimage/bin/java -listmods
```

```
java.base@9.0
java.logging@9.0
java.sql@9.0
java.xml@9.0
com.azul.app@1.0
com.azul.zoop@1.0
com.azul.zeta@1.0
```

Version numbering for information purposes only  
“It is not a goal of the module system to solve the version-selection problem”

# Multi Release jar

---

```
multirelease.jar  
├── META-INF  
│   └── versions  
│       └── 9  
│           └── multirelease  
│               └── Helper.class  
└── multirelease  
    ├── Helper.class  
    └── Main.class
```

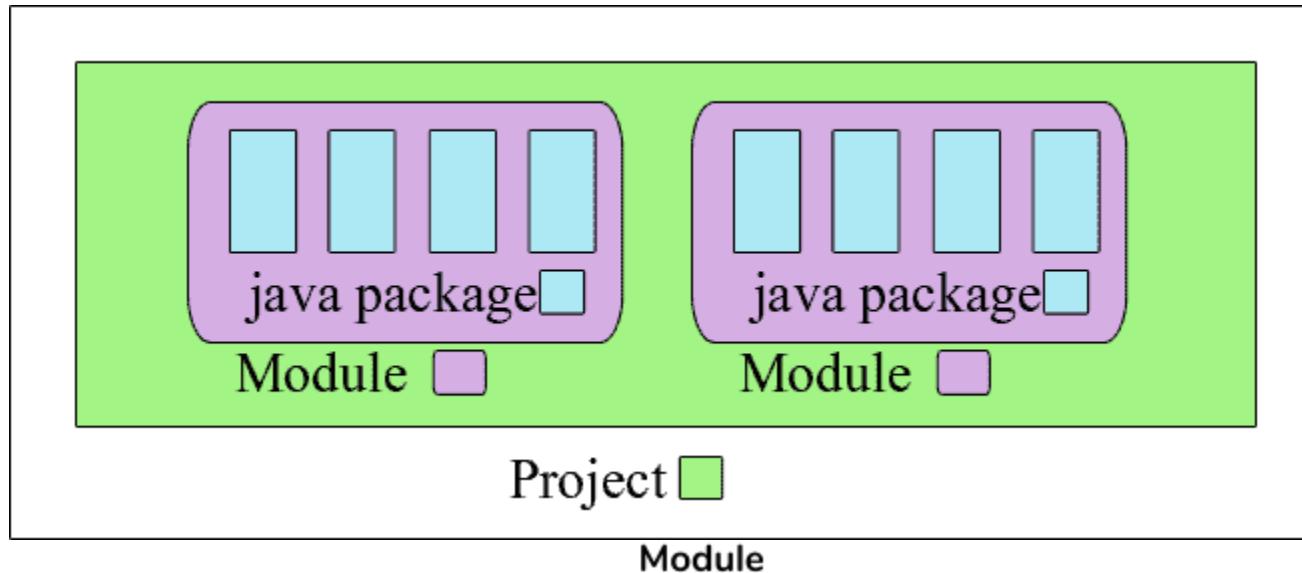


## Features of java 9 Modules

---

- Increases code reusability: by creating modules we can use them in different projects
- Easy and meaningful grouping of packages: if we have many packages in one project it is difficult to manage and organize code, this is where modules come to the rescue
- More abstraction to packages: we can decide which packages are allowed to be accessed outside and which are private or for internal use
- Separation of resource: each module will have its own required resource files like media or configuration files
- Internal or secure classes can be hidden from outside world

# Features of java 9 Modules



## Exported Packages -

These packages can be used outside of this module

## Concealed Packages -

These packages cannot be used outside of this module.  
Only the code of this module can use these packages.

# Features of java 9 Modules

## java.base

### Exported Packages

java.io  
java.time  
java.util  
java.net  
java.lang

### Concealed Packages

sun.security.provider

...

...

```
// file name: module-info.java
module java.base {
```

```
    exports java.io;
    exports java.time;
    exports java.util;
    exports java.net;
    exports java.lang;
```

```
}
```

## Steps to create Module

---

- Create a folder with module name. Generally company name in reverse with artifact name is used. eg: ‘com.stacktraceguru.util’
- Add file with name ‘module-info.java’ in module root folder. This file is called as ‘Module Descriptor’ file
- Create java packages as per requirement
- Add classes as required under the created packages

# Create module

---

```
Microsoft Windows [Version 10.0.22621.2134]
(c) Microsoft Corporation. All rights reserved.

I:\hsbc2023projects\BankingApp>javac -d mods/com.hsbc.banking.utilities src/module-info.java src/com/hsbc/banking/utilities/App.java
I:\hsbc2023projects\BankingApp>
```

```
javac -d mods/com.hsbc.banking.utilities src/module-info.java
src/com/hsbc/banking/utilities/App.java
```

## Multi-Release JAR Files

---

- Multi-release JARs (MR-JARs) are specially prepared JARs that contain bytecode for several major Java versions, where...
- Java 8 and older load version-unspecific class files
- Java 9 and newer load version-specific class files if they exist, otherwise falling back to version-unspecific ones

## What are the rules for creating Module?

---

- Module name must be unique
- Each module must have exactly one Module Descriptor file with name ‘module-info.java’
- Package names must be unique. Even in the different modules we cannot have same package names
- We can add media and other resource files in the module
- Each module will create one jar file. For multiple jars we need to create separate modules
- One project can have multiple modules

# What are the Module types?

---

- Depending on how the modules are used, they are categorized into 4 types,
  - System Modules: the modules from JDK and JRE. Can be listed using `java --list-modules`
  - Application Modules: all the modules created in an application to achieve a functionality
  - Automatic Modules: existing jar files which are not modules but are added to module path. When we add non module jars to module path, module with jar name is created.
    - By default exports all the packages
    - By default can access classes from all other modules

## What are the Module types?

---

- Unnamed Module: jars and classes added into the classpath. When we add jar or class to the classpath all these classes are added to the unnamed module
- Only exports to other unnamed module and automatic module. This means, application modules cannot access these classes
- It can access classes from all the modules

## What is Module Descriptor file?

---

- It is a file with name `module-info.java`, under the root module path. This file contains the module metadata information.
- This is also java file which is compileable using `javac` command.

## This file defines following things

---

- Public packages: list of packages that current module exports using ‘exports’ keyword
- Dependencies on other modules: list of other modules on which the current module is dependent on. This is done using ‘requires’ keyword
- Services offered: list of services that current module provides using ‘provides’ keyword
- Services consumed: list of services that current module consumes using ‘uses’ keyword
- Reflection permission: permission to specify if reflection can be used to access private members using ‘open’ keyword

## This file defines following things

---

```
module com.module.util{ // module <module.name>
    exports com.module.util;
    requires java.sql;
}
```

# Exports

---

- By default all the packages are private and we can make them public using exports keyword.
- `exports <packageToExport>;`
- `module com.module.util{`
- `exports com.module.package1;`
- `exports com.module.package2;`
- `}`

## Rules to use export keyword:

---

- only exports packages not classes
- each package requires new exports keyword
- `module com.module.util{`
- `exports com.module.package1;`
- `exports com.module.package2 to com.module.app;`
- `exports com.module.package3 to com.module.app,`  
 `com.module.help;`
- `}`

# Requires

---

- If a module needs to access packages exported from other modules, then these other modules must be imported using ‘requires’ keyword.
- Only after specifying the module dependency using ‘requires’, the other module packages can be used.
- `requires <module-to-access>;`
- `module com.module.app{`
- `requires java.sql;`
- `requires com.module.util;`
- `}`

## Rules to use requires keyword:

---

- only module can be specified for ‘requires’. Packages cannot be specified
- dependency of each module must be specified separately, with separate ‘requires’ keyword

## Requires Static

---

- Sometimes we need some modules during compile time only and they are optional at runtime. For example, testing or code generation libraries.
- If we need compile time dependency that is optional at runtime then this dependency must be specified using ‘requires static’ keyword.
- `requires static <module-to-access>;`
- `module com.module.app{`
- `requires static java.sql;`
- `requires com.module.util;`
- `}`

# Requires Transitive

---

- There is a possibility to grant access of the modules, on which our current module depends, to the module that uses our current module. The ‘requires transitive’ keyword helps to achieve this.
- This means all the modules that are using our module will get the access to transitive dependency automatically.
- Syntax
- **requires transitive <module-to-access>;**
- Example
- ```
module com.module.app{  
    requires transitive com.module.util;  
    requires java.sql;  
}
```

## Uses

---

- Using uses keyword we can specify that our module needs or consumes some service. Service is a interface or abstract class. It should not be an implementation class.
- Syntax
- **uses <service-required>;**
- Example
- **module com.module.util{**
- **uses com.util.PersonDataService;**
- **}**

## Provides ... With

---

- We can specify that our module provides some services that other modules can use.
- Syntax
- `provides <service-provided> with <service-implementation-class> ;`
- Example
- `module com.module.util{`
- `provides com.util.PersonDataService with`  
 `com.util.DbPersonServiceImpl;`
- `}`

# Open

---

- Since java 9 encapsulation and security is improved for the reflection apis. Using reflection we were able to access even the private members of the objects.
- From java 9 this is not open by default. We can although grant reflection permission to other modules explicitly.
- `open module com.module.util{`
- `}`

## Opens

---

- If we do not want to open all the packages for reflection we can specify packages manually using ‘opens’ keyword.
- module com.module.util{
  - opens com.module.package1;
  - }

## Opens .. To

---

- Using ‘opens ...to’ keyword we can open reflection permission for specific packages to specific modules only.
- module com.module.util{
  - opens com.module.package1 to module.a, module.b, org.test.integration;
  - }
- In this case only module.a, module.b, org.test.integration modules can access classes from package1 using reflection.

## Aggregator Module

---

- First of all, this is not a technical concept. It is just a convenience concept for developers to make their life easier.
- Sometimes multiple modules require other multiple modules.
- Instead of adding these in every module descriptor, we can create one module that will add all the required dependency using ‘transitive’.
- Then we just need to add dependency of this module wherever needed, this will add all the required modules transitive dependency.
- This common module is the ‘Aggregator module’.

# Aggregator Module

---

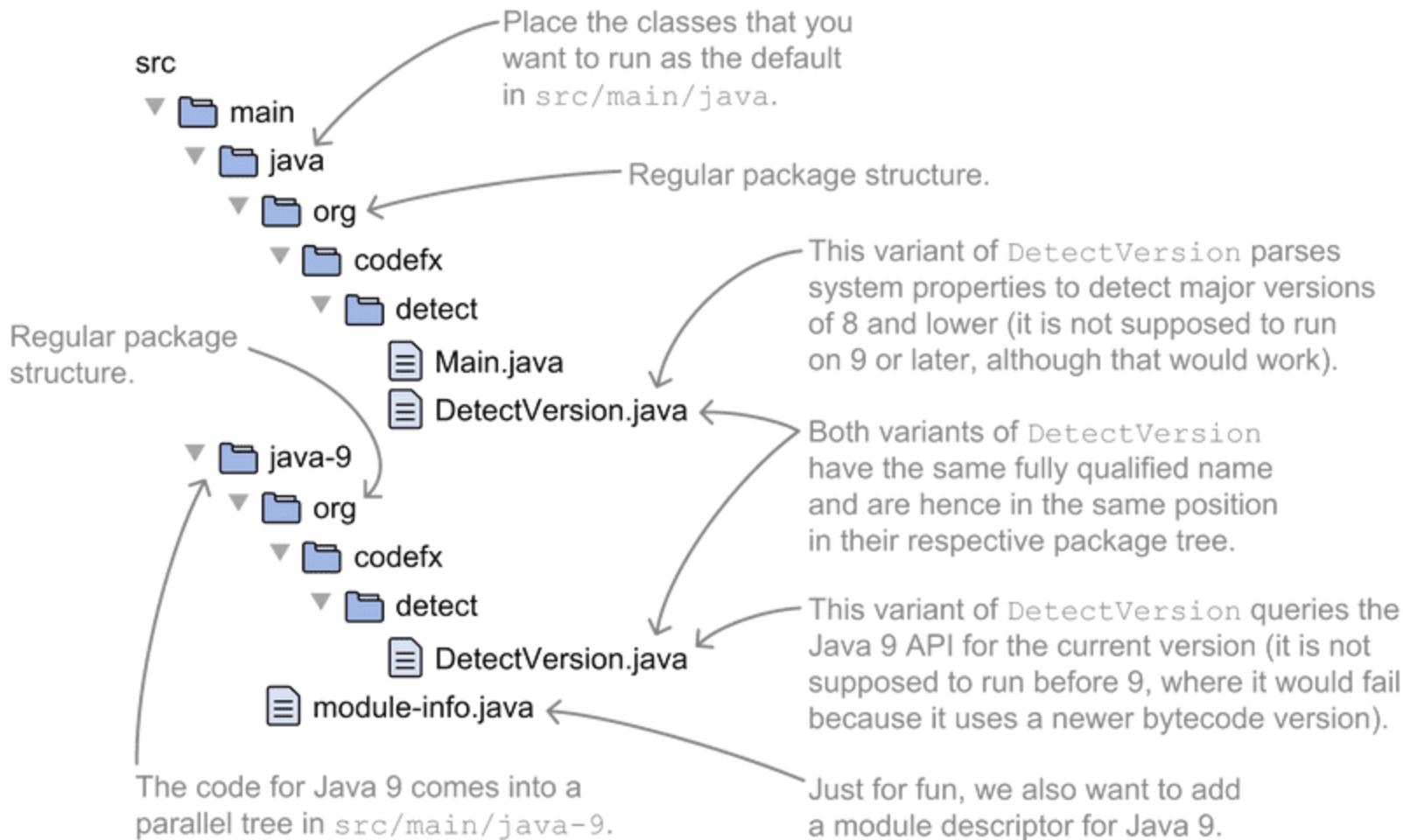
- For example, we have 10 modules, modA to modJ. modP, modQ, modR needs all 10 modules, then we can create one common module as below,
- module modulePQR{
  - requires transitive modA;
  - ....
  - ...
  - requires transitive modJ;
- }
- Then modules P, Q and R just need to add require for modulePQR
- module modP{
  - requires transitive modulePQR;
- }

## Multi-Release JAR Files

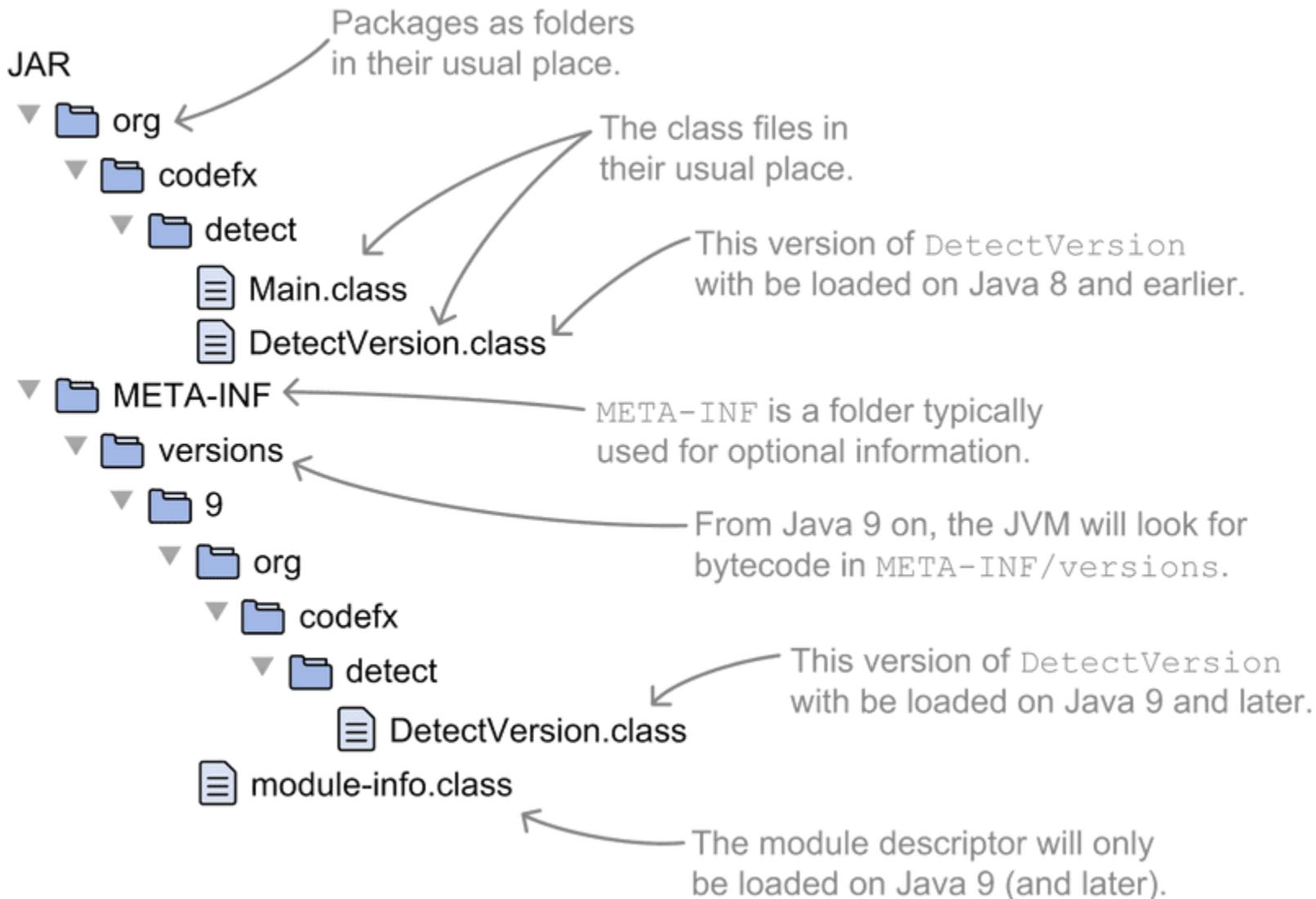
---

- The solution in Java 9 is to leave the original class untouched and instead create a new version using the new JDK and package them together.
- At runtime, the JVM (version 9 or above) will call any one of these two versions giving more preference to the highest version that the JVM supports.
- For example, if an MRJAR contains Java version 7 (default), 9 and 10 of the same class, then JVM 10+ would execute version 10, and JVM 9 would execute version 9.
- In both cases, the default version is not executed as a more appropriate version exists for that JVM.

# Multi-Release JAR Files



# Internal Workings Of Multi-release JARs



# Multi-Release JAR Files

---

- how to compile and package them into an MR-JAR:
- # compile code in `src/main/java` for Java 8 into `classes`
  - \$ javac --release 8
    - -d classes
    - src/main/java/org/codefx/detect/\*.java
- # compile code in `src/main/java-9` for Java 9 into `classes-9`
  - \$ javac --release 9
    - -d classes-9
    - src/main/java-9/module-info.java
    - src/main/java-9/org/codefx/detect/DetectVersion.java
- # when packaging the bytecode into a JAR, the first part (up to
- # `-C classes .`) packages "default" bytecode from `classes` as
- # usual; the new bit is the `--release 9` option, followed by

# Multi-Release JAR Files

---

```
I:\hsbc2023projects\multireleasejar>javac --release 7 -d classes src\main\java\com\hsbc\multireleasejar\*.java  
I:\hsbc2023projects\multireleasejar>javac --release 9 -d classes-9 src\main\java9\com\hsbc\multireleasejar9\*.java  
I:\hsbc2023projects\multireleasejar>
```

```
javac --release 9 -d classes-9 src\main\java9\com\hsbc\multireleasejar9\*.java
```

# Multi-Release JAR Files

```
F:\java9ws\simpleexercises>javac --release 9 -d mods/com.boa.simpleexercises src/main/java/module-info.java src/main/java/com/boa/simpleexercises/SafeArgsTest.java
Note: src\main\java\com\boa\simpleexercises\SafeArgsTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

F:\java9ws\simpleexercises>
```

```
F:\java9ws\simpleexercises>javac --release 9 -d classes-9 src/main/java/module-info.java src/main/java/com/boa/simpleexercises/SafeArgsTest.java
Note: src\main\java\com\boa\simpleexercises\SafeArgsTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

F:\java9ws\simpleexercises>■
```

# Module-info

```
F:\java9ws\simpleexercises>javac --release 9 -d mods/com.boa.simpleexercises src/main/java/module-info.java src/main/java/com/boa/simpleexercises/SafeArgsTest.java
Note: src\main\java\com\boa\simpleexercises\SafeArgsTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

F:\java9ws\simpleexercises>
```

## Multi-Release JAR Files

---

- # more classes to include specifically for Java 9
- \$ jar --create
- --file target/detect.jar
  - -C classes .
  - --release 9
  - -C classes-9 .
- jar --create --file target/detect.jar -C classes . --release 9 -C classes-9 .
- jar cvf target/safetest.jar -C classes-9 .
- By running the resulting JAR on JVMs version 8 and 9, you can observe that, depending on the version, a different class is loaded

# Multi-Release JAR Files

```
Administrator: Command Prompt

F:\java9ws\simpleexercisesmrjars>jar cvf target/safetest.jar -C classes-9 .
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/boa/(in = 0) (out= 0)(stored 0%)
adding: com/boa/simpleexercises/(in = 0) (out= 0)(stored 0%)
adding: com/boa/simpleexercises/SafeArgsTest.class(in = 817) (out= 543)(deflated 33%)
adding: module-info.class(in = 189) (out= 136)(deflated 28%)

F:\java9ws\simpleexercisesmrjars>
```

```
I:\hsbc2023projects\multireleasejar>jar cvf target/mrjar.jar -C classes-9 .
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/hsbc/(in = 0) (out= 0)(stored 0%)
adding: com/hsbc/multireleasejar9/(in = 0) (out= 0)(stored 0%)
adding: com/hsbc/multireleasejar9/App.class(in = 1184) (out= 658)(deflated 44%)
adding: com/hsbc/multireleasejar9/DataHelper.class(in = 671) (out= 437)(deflated 34%)

I:\hsbc2023projects\multireleasejar>|
```

## Stream API Implementation

---

- Java 9 added the following four methods to the Stream.
- Since Stream is an interface, the methods added to it are default and static.

# Stream API Implementation

---

- **Java 9 – Stream dropWhile() method**
- The method dropWhile() drops all the elements of the stream until the given predicate fails.
- For example:

```
jshell> Stream<Integer> mystream = Stream.of(11, 22, 40, 60, 100)
mystream ==> java.util.stream.ReferencePipeline$Head@3159c4b8
```

```
jshell> mystream.dropWhile(num -> num < 50).forEach(num ->
System.out.println(num))
60
100
```

# Stream API Implementation

---

- **Java 9 – Stream takeWhile() method**
- The method takeWhile() works just opposite to the dropWhile() method.
- This method takes all the elements of the stream in the resulted stream until the predicate fails.
- In short, when the predicate fails, it drops that element and all the elements that comes after that element in the stream. Lets take few examples to understand this.
- Using takeWhile() method on an ordered Stream
- Here the stream is ordered and the takeWhile() method takes all the elements until the predicate fails at element value 60.

# Stream API Implementation

---

```
jshell> Stream<Integer> mystream = Stream.of(10, 20, 30, 40, 60, 90, 120)
mystream ==> java.util.stream.ReferencePipeline$Head@2038ae61
jshell> mystream.takeWhile(num -> num < 50).forEach(num ->
System.out.println(num))
10
20
30
40
```

# Stream API Implementation

---

- **Java 9 – Stream iterate() method**
- The iterate method in Java 9 has three arguments.
- First argument is the initialising value, the returned stream starts with this value.
- Second argument is the predicate, the iteration continues until this given predicate returns false.
- Third argument updates the value of previous iteration.
- Example:
- In this example, the first argument is 1. The stream starts with the element 1.
- num -> num < 30 is the second argument and it is a predicate. The iteration continues until this returns false. num -> num\*3 is the third argument that updates the value returned from previous iteration. This works similar to the counter variable of a loop.
- ```
jshell> IntStream.iterate(1, num -> num < 30, num -> num*3).forEach(num ->System.out.println(num))
```
- 1
- 3
- 9
- 27

# Stream API Implementation

---

- **Java 9 – Stream ofNullable() method**
- This method is introduced to avoid NullPointerException. This method returns an empty stream if the stream is null. It can also be used on a non-empty stream where it returns a sequential stream of single element.
- Null Stream Example
- `jshell> Stream<String> stream = Stream.ofNullable(null)`
- `stream ==> java.util.stream.ReferencePipeline$Head@2286778`
- `jshell> stream.forEach(str -> System.out.println(str))`
- `jshell>`

# Stream API Implementation

---

- **Non-null Stream Example**
- jshell> Stream<String> stream = Stream.ofNullable("Rose")
- stream ==> java.util.stream.ReferencePipeline\$Head@370736d9
- jshell> stream.forEach(str -> System.out.println(str))
- Rose
- jshell>

## Filter vs dropwhile() vs takeWhile()

---

- Streams have a filter operation that filters out the elements that do not match the predicate.
- But it processes the entire stream.
- On the other hand, takeWhile and dropWhile take or drop a contiguous or sequence of elements.
- In other words, for takeWhile, there may be elements that will satisfy the predicate in the discarded remaining portion of the stream.
- Similarly, for dropWhile, there may be elements that fail to satisfy the condition in the selected portion of the stream.

## Immutable Collection

---

- 1. Immutable collections in java are those collections that can't be modified once they are created.
- An instance/object of immutable collection holds the same data as long as a reference to it exists, you can't modify it after creation.
- 2. Immutable collections are thread-safe because any thread can't modify it.
- So multiple threads can read it but can't modify it because the structure doesn't support the mutation.

# Immutable Collection

Before Java 9	After Java 9
<b>1) Immutable List :</b> <pre>List&lt;String&gt; sportList = new ArrayList&lt;String&gt;();  sportList.add("Hockey"); sportList.add("Cricket"); sportList.add("Tennis");  List&lt;String&gt; unModifiableSportList = Collections.unmodifiableList(sportList);</pre>	<pre>List&lt;String&gt; immutableSportList = List.of("Hockey", "Cricket", "Tennis");</pre>
<b>2) Immutable Set :</b> <pre>Set&lt;String&gt; sportSet = new HashSet&lt;&gt;();  sportSet.add("Hockey"); sportSet.add("Cricket"); sportSet.add("Tennis");  Set&lt;String&gt; unModifiableSportSet = Collections.unmodifiableSet(sportSet);</pre>	<pre>Set&lt;String&gt; immutableSportSet = Set.of("Hockey", "Cricket", "Tennis");</pre>
<b>3) Immutable Map :</b> <pre>Map&lt;Integer, String&gt; sportMap = new HashMap&lt;&gt;();  sportMap.put(1, "Hockey"); sportMap.put(2, "Cricket"); sportMap.put(3, "Tennis");  Map&lt;Integer, String&gt; unModifiableSportMap = Collections.unmodifiableMap(sportMap);</pre>	<pre>Map&lt;Integer, String&gt; immutableSportMap = Map.of(1, "Hockey", 2, "Cricket", 3, "Tennis");</pre>

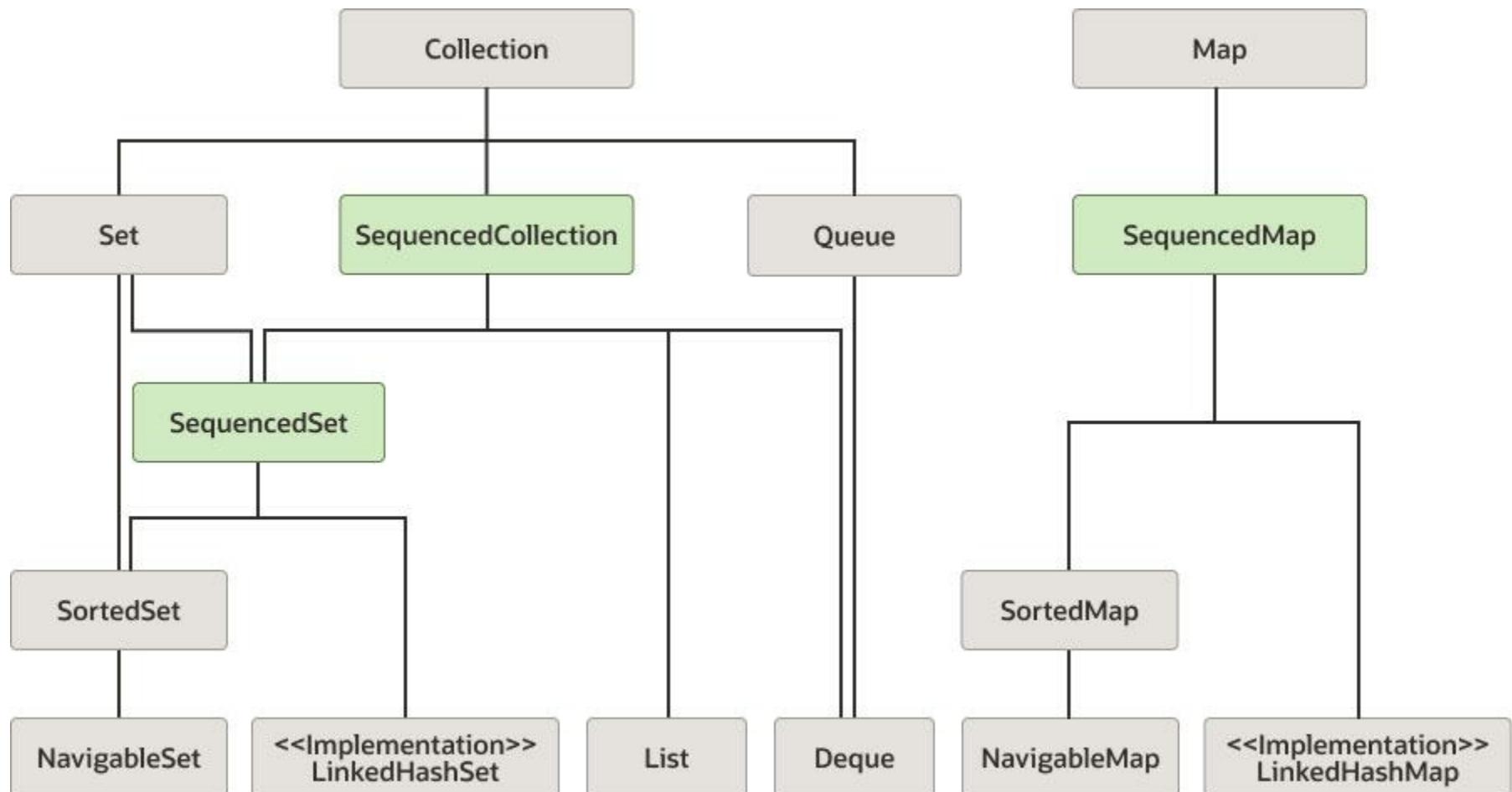
# Immutable Copy of List , Set and Map Java 10

```
*CopyOfApiRunner.java  List.class  Set.class  Map.class
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class CopyOfApiRunner {
7
8     public static void main(String[] args) {
9         List<String> names = new ArrayList<String>();
10        names.add("Ranga");
11        names.add("Ravi");
12        names.add("John");
13
14        List<String> copyOfNames = List.copyOf(names);
15
16        doNotChange(copyOfNames);
17        System.out.println(copyOfNames);
18
19    }
20
21    private static void doNotChange(List<String> names) {
22        names.add("ShouldNotbeAllowed");
23    }
24}
```

Not  
Allowed in  
Immutable



# Sequenced Collection in java 21



# Sequenced Collection in java 21

Operations	SequencedCollection	SortedSet	List	LinkedHashSet
Get the first element	getFirst()	✓	✓	✓
Remove the first element	removeFirst()	✓	✓	✓
Inserting an element at first	addFirst()	Not supported <sup>(1)</sup>	✓	✓ <sup>(2)</sup>
Get the last element	getLast()	✓	✓	✓
Remove the last element	removeLast()	✓	✓	✓
Inserting an element at last	addLast()	Not supported <sup>(1)</sup>	✓	✓ <sup>(2)</sup>

# Sequenced Collection in java 21

---

```
SequencedSet<String> sequencedSet = new LinkedHashSet<>();  
sequencedSet.addFirst("Apple");  
sequencedSet.add("Banana");  
sequencedSet.addLast("Cherry");
```

```
SequencedSet<String> sequencedSet = new LinkedHashSet<>();  
sequencedSet.addFirst("Apple");  
sequencedSet.add("Banana");  
sequencedSet.addLast("Cherry");  
  
assertEquals("Apple", sequencedSet.getFirst());  
assertEquals("Cherry", sequencedSet.getLast());
```

# Sequenced Collection in java 21

---

```
SequencedMap<String, Integer> sequencedMap = new LinkedHashMap<>();
sequencedMap.putFirst("Apple", 10);
sequencedMap.put("Banana", 20);
sequencedMap.putLast("Cherry", 30);

assertEquals("Apple", sequencedMap.firstEntry().getKey());
assertEquals(10, sequencedMap.firstEntry().getValue());

assertEquals("Cherry", sequencedMap.lastEntry().getKey());
assertEquals(30, sequencedMap.lastEntry().getValue());
```

## Pattern Matching Instance of Operator

---

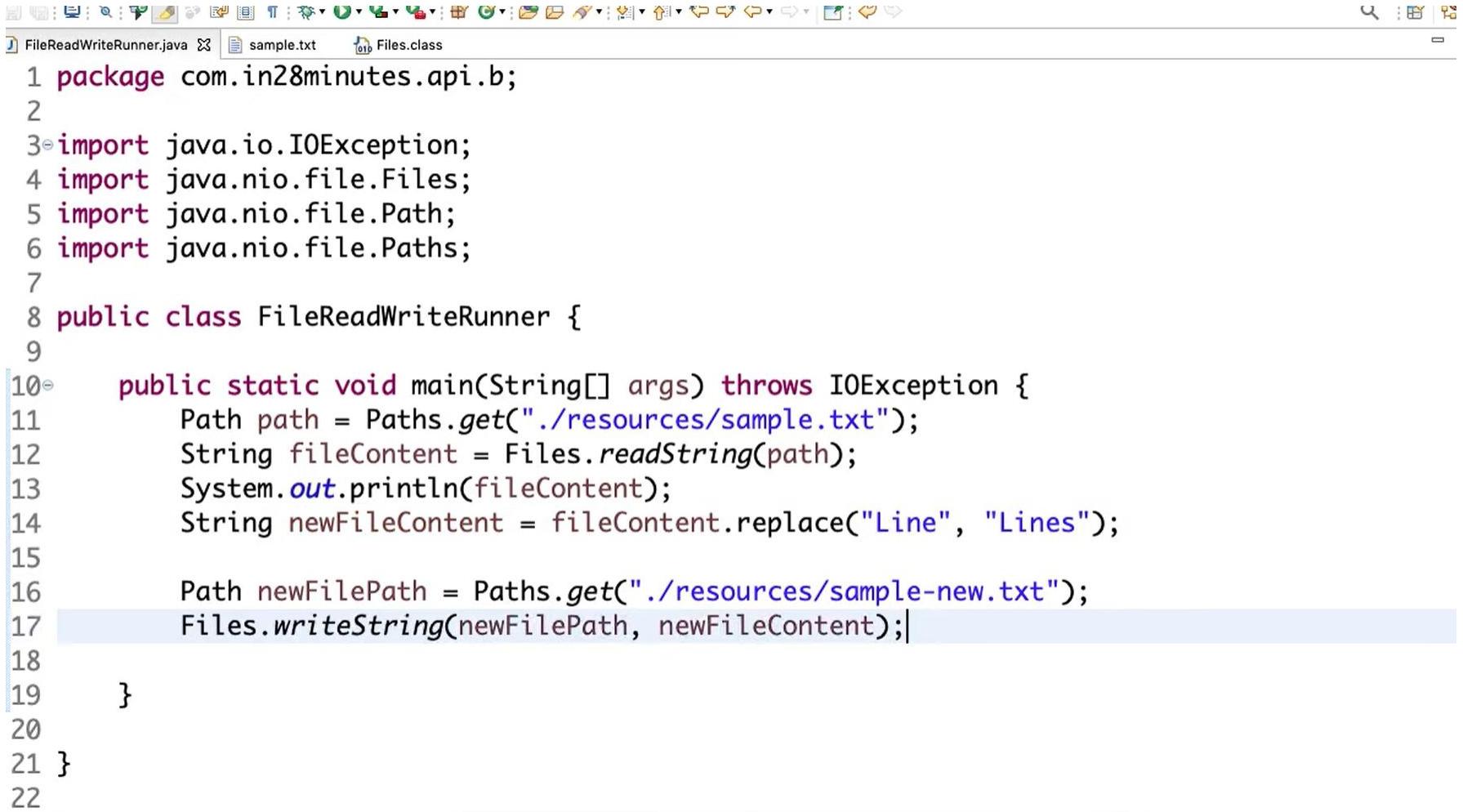
- Java 14, via JEP 305, brings an enhanced version of the instance Of keyword that both tests the parameter object and assigns it to a binding variable of the correct type.
- Scope of pattern variables:
- Note that the assignment of the pattern variable happens only when the predicate test is true.
- The variable is available in the enclosing if block, and we cannot access it outside it.

# Pattern Matching Instance of Operator

---

```
public class GFG {  
    public static void resolveTypeOfObject(Animal animal)  
    {  
        if (animal instanceof Cat cat) {  
            cat.meow();  
            // other cat operations  
        }  
        else if (animal instanceof Dog dog) {  
            dog.woof();  
            // other dog operations  
        }  
    }  
  
    public static void main(String[] args)  
    {  
        Animal animal = new Dog();  
        resolveTypeOfObject(animal);  
  
        animal = new Cat();  
        resolveTypeOfObject(animal);  
    }  
}
```

# File Read Write Enhancements



The screenshot shows a Java code editor with the following code:

```
1 package com.in28minutes.api.b;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7
8 public class FileReadWriteRunner {
9
10    public static void main(String[] args) throws IOException {
11        Path path = Paths.get("./resources/sample.txt");
12        String fileContent = Files.readString(path);
13        System.out.println(fileContent);
14        String newFileContent = fileContent.replace("Line", "Lines");
15
16        Path newPath = Paths.get("./resources/sample-new.txt");
17        Files.writeString(newPath, newFileContent);
18
19    }
20
21 }
22
```

The code demonstrates reading the content of a file named 'sample.txt' located in the resources directory and writing its content to a new file named 'sample-new.txt'. The content is modified by replacing the word 'Line' with 'Lines'.

# Predicate Enhancements

The screenshot shows the Eclipse IDE interface with the title bar "java-9-course-for-beginners-Jan18 - java-new-api-features/src/com/in28minutes/api/c/PredicateNotRunner.java - Eclipse IDE". The code editor displays the following Java code:

```
1 package com.in28minutes.api.c;
2
3 import java.util.List;
4 import java.util.function.Predicate;
5
6 public class PredicateNotRunner {
7
8     public static boolean isEven(Integer number) {
9         return number%2==0;
10    }
11
12    public static void main(String[] args) {
13        List<Integer> numbers = List.of(3,4,5,67,89,88);
14        // Predicate<Integer> evenNumberPredicate = number -> number%2==0;
15        // numbers.stream().filter(evenNumberPredicate.negate()).forEach(System.out::println);
16        numbers.stream().filter(Predicate.not(PredicateNotRunner::isEven))
17            .forEach(System.out::p
18
19
20
21    }
22}
```

A tooltip is visible over the line "numbers.stream().filter(Predicate.not(PredicateNotRunner::isEven))" with the following options:

- Open Declaration
- Open Implementation
- Open Return Type

The status bar at the bottom shows "Writable", "Smart Insert", "21 : 6 : 570", and a play button icon.

## JAVA 9 CONCURRENCY UPDATES

---

- `java.util.concurrent.Flow` has been introduced, which has nested interfaces supporting the implementation of a publish-subscribe framework.
- The publish-subscribe framework enables developers to build components that can asynchronously consume a live stream of data by setting up publishers that produce the data and subscribers that consume the data via subscription, which manages them.

# JAVA 9 CONCURRENCY UPDATES

---

- The four new interfaces are as follows:
  - `java.util.concurrent.Flow.Publisher`
  - `java.util.concurrent.Flow.Subscriber`
  - `java.util.concurrent.Flow.Subscription`
  - `java.util.concurrent.Flow.Processor` (which acts as both Publisher and Subscriber).

## Publish-subscribe framework for reactive streams

---

- Data processing has evolved from batch architectures that collect data and subsequently process the data after some threshold has been reached.
- To stream-oriented architectures that help to turn data into knowledge as quickly as possible.
- Stream-oriented architectures capture and process live data, and modify systems based on the processed results very quickly (typically in seconds or less).

## Publish-subscribe framework for reactive streams

---

- Handling streams of data (especially "live" data whose volume isn't predetermined) requires special care in an asynchronous system.
- The main issue is that resource consumption needs to be controlled so that a fast data source doesn't overwhelm the stream destination.
- Asynchrony is needed to enable the parallel use of computing resources, on collaborating network hosts or multiple CPU cores within a single machine, which can greatly speed up data processing.

# Publish-subscribe framework for reactive streams

---

- The Reactive Streams initiative provides a standard for asynchronous stream processing with nonblocking back pressure.
- A reactive stream provides a way to signal its source to ease production of data when the stream's destination becomes overwhelmed with that data.
- This signaling capability is like a valve on a water pipe.
- Closing this valve increases back pressure (the pressure back at the source) while easing the burden on the destination.

## Publish-subscribe framework for reactive streams

---

- The initiative's objective is to govern the exchange of stream data across an asynchronous boundary (such as passing data to another thread) while ensuring that the destination isn't forced to buffer arbitrary amounts of data.
- In other words, back pressure is an integral part of this model in order to allow the queues that mediate between threads to be bounded.
- Note that the communication of back pressure is handled asynchronously.

## Publish-subscribe framework for reactive streams

---

- Reactive Streams focuses on finding a minimal set of interfaces, methods, and protocols for describing the operations and entities needed to achieve the objective: asynchronous streams of data with nonblocking back pressure.

# Exploring the publish-subscribe framework

---

- Java 9 supports the Reactive Streams initiative by providing a publish-subscribe framework (also known as the Flow API) that consists of the **java.util.concurrent.Flow** and **java.util.concurrent.SubmissionPublisher** classes.
- Flow is a repository for four nested static interfaces whose methods establish flow-controlled components in which publishers produce data items that are consumed by one or more subscribers

# Exploring the publish-subscribe framework

---

- Publisher: A producer of data items that are received by subscribers.
- Subscriber: A receiver of data items.
- Subscription: Linkage between a Publisher and a Subscriber.
- Processor: A combination of Publisher and Subscriber for specifying a data-transformation function.

# Exploring the publish-subscribe framework

---

- A publisher publishes a stream of data items to registered subscribers and implements Flow.Publisher.
- This interface declares a single method, which is invoked to register a subscriber with a publisher:
- `void subscribe(Flow.Subscriber<? super T> subscriber)`
- Invoking this method registers subscriber with the publisher.
- However, if subscriber is already registered or the registration fails due to some policy (or other) violation, this method invokes subscriber's `onError()` method with an `IllegalStateException` object.

## Exploring the publish-subscribe framework

---

- Otherwise, subscriber's `onSubscribe()` method is invoked with a new `Flow.Subscription` object.
- `subscribe()` throws `NullPointerException` when null is passed to subscriber.

# Exploring the publish-subscribe framework

---

- A subscriber subscribes to a publisher for callbacks of data items and implements Flow.Subscriber<T>.
- This interface declares onSubscribe() and three additional methods:
  - void onSubscribe(Flow.Subscription subscription)
  - void onComplete()
  - void onError(Throwable throwable)
  - void onNext(T item)

# Exploring the publish-subscribe framework

---

- `onSubscribe()` is invoked to confirm registration. It receives a `Subscription` argument whose methods allow requests for new data items to be made to the publisher or to request that the publisher send no more data items.
- `onComplete()` is invoked when it's known that no additional `Subscriber` method invocations will occur for a `Subscription` that's not already terminated by error. No other `Subscriber` methods are called after this method.

# Exploring the publish-subscribe framework

---

- `onError(Throwable throwable)` is invoked with the specified `throwable` upon an unrecoverable error that's encountered by the publisher or subscription. No other `Subscriber` methods are called after this method.
- `onNext()` is invoked with a subscription's next item. If this method throws an exception, the resulting behavior isn't guaranteed, but may cause the subscription to be cancelled.

## Exploring the publish-subscribe framework

---

- A subscription provides a link between a publisher and a subscriber, letting subscribers receive data items only upon request and cancel at any time.
- Subscriptions implement the `Flow.Subscription` interface, which declares two methods:
  - `void request(long n)`
  - `void cancel()`

# Exploring the publish-subscribe framework

---

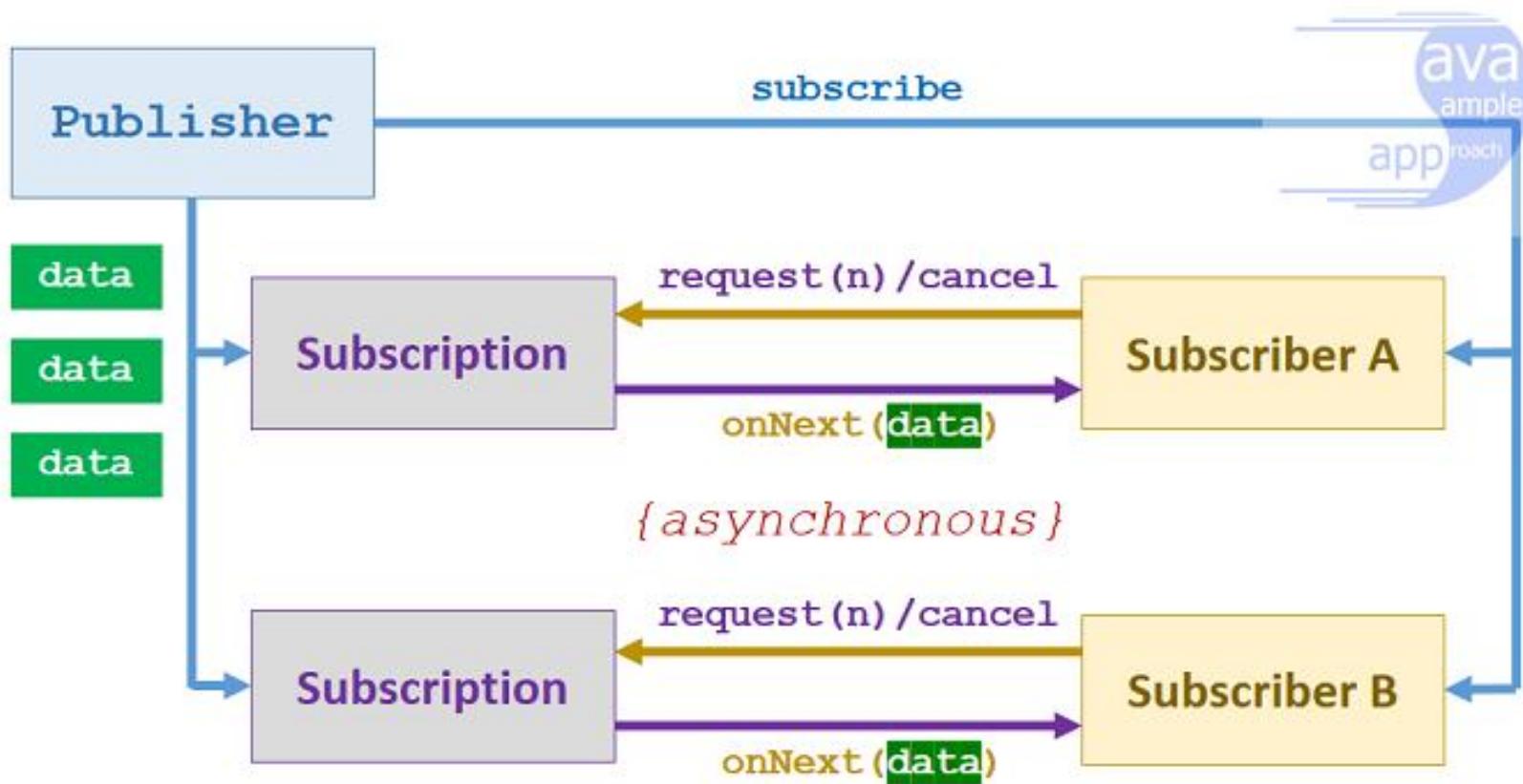
- `request()` adds n data items to the current unfulfilled demand for this subscription.
- If n is less than or equal to 0, the subscriber's `onError()` method is called with an `IllegalArgumentException` argument.
- Otherwise, the subscriber receives up to n additional `onNext()` invocations (or fewer when terminated). Passing `Long.MAX_VALUE` to n indicates an unbounded number of invocations.

# Exploring the publish-subscribe framework

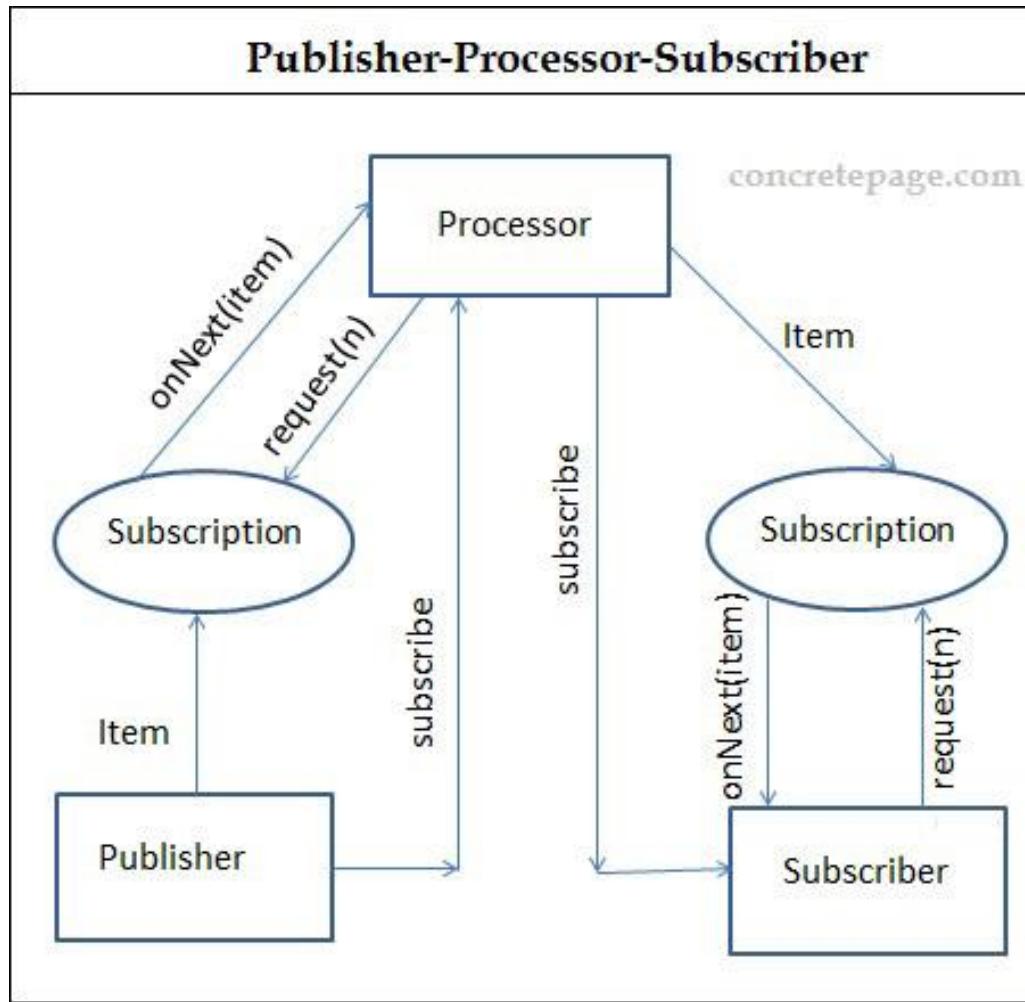
---

- `cancel()` causes the subscriber to eventually stop receiving data items. A best-effort attempt is made; additional data items may be received after `cancel()` is called.

# Exploring the publish-subscribe framework



# Exploring the publish-subscribe framework



# CompletableFuture enhancements

---

- Java 8 introduced the `CompletableFuture<T>` class, which is a `java.util.concurrent.Future<T>` that may be explicitly completed (setting its value and status), and may be used as a `java.util.concurrent.CompletionStage`, supporting dependent functions and actions that are triggered upon the future's completion.
- Java 9 introduces several enhancements to `CompletableFuture`:
  - support for delays and timeouts
  - improved support for subclassing
  - new factory methods

# Completable Future improvements

---

- defaultExecutor method
- copy method
- completeAsync method
- orTimeout method
- completeOnTimeout method
- New Factory methods

## defaultExecutor()

---

- Executor executor = new CompletableFuture<>().defaultExecutor();
- CompletableFuture nameFuture = CompletableFuture.supplyAsync(() -> "Deepak", executor);
- // Should be avoided, only for demonstration purpose
- System.out.println(nameFuture.join());

## copy()

---

- returns an immutable copy of the existing CompletableFuture instance, since the copy is immutable it would not affect the original instance in any way.
- CompletableFuture nameFuture = CompletableFuture.supplyAsync(() -> "Deepak");
- CompletableFuture nameFutureCopy = nameFuture.copy();
- CompletableFuture withSurname = nameFuture.thenApply(name -> "Deepak Mehra");

## copy()

---

- // Should be avoided, only for demonstration purpose
  - System.out.println(withSurname.join());
- 
- // Change in original instance should affect it's copy instance.
  - System.out.println(nameFutureCopy.join());

## completeAsync()

---

- Completes this CompletableFuture with the result of the given Supplier function invoked from an asynchronous task using the default executor.
- CompletableFuture completableFuture = new CompletableFuture();
- CompletableFuture nameFuture = completableFuture.completeAsync(() -> "Deepak");
- // Should be avoided, only for demonstration purpose
- System.out.println(nameFuture.join());

## orTimeout

---

- Prior to orTimeout() what we had, was the get() method where we can define the Time Units and if the future call is not completed in that particular time, it will raise the timeout exception.
- But this method will block the thread and as a result it will not be non-blocking anymore.
- In order to deal with this, Java9 introduced orTimeout() which will provide us the timeout functionality.
- Keep the calls non-blocking at the same time i.e the thread will not be blocked for that point of time and you can still have the timeout feature for your future calls.

## completeOnTimeout

---

- This method is essentially similar to orTimeout().
- However, with this method you can also provide a default or static value if the service is taking too long to respond or there is a TimeoutException.
- This works more like a method which is capable of returning a fallback value when something goes wrong.

## New Static Factory Methods

---

- completedFuture() – Returns a new CompletableFuture that is already completed with the given value.
- completedStage() – Returns a new CompletionStage that is already completed with the given value and supports only those methods in interface CompletionStage.
- failedStage() – Returns a new CompletionStage that is already completed exceptionally with the given exception and supports only those methods in interface CompletionStage
- completedFuture() signature – public static CompletableFuture completedFuture(U value)

## New Static Factory Methods

---

- completedStage() signature – public static CompletionStage completedStage(U value)
- failedStage() signature – public static CompletionStage failedStage(Throwable ex)

# Optional Enhancements

---

- Stream()
- ifPresentorElse()
- or

## StackWalker API In Java 9

---

- Java 9 introduces an efficient API for obtaining stack trace information.
- It allows easy filtering and lazy access to the execution stacks (also called stack frames) within any method.

## StackWalker

---

- The class `java.lang.StackWalker`, is the main class we are going to work with. With this class we can do following things:
- Traversing the StackFrames by using `StackWalker.forEach()` method.
- Walk through the stack traces and applying a function (filtering and mapping etc) on a Stream of StackFrames by using `StackWalker.walk()` method.
- Getting the caller class in an efficient way by calling `StackWalker.getCallerClass()`

## StackWalker.StackFrame

---

- This class is static nested class of StackWalker.
- It represents a method invocation return by StackWalker.
- It has methods to access a given stack frame information i.e. getDeclaringClass(), getLineNumber() etc (equivalent to what we have in StackTraceElement class).

## StackWalker.Option

---

- This is also a static nested class of StackWalker.
- It provides options for Stack walker to configure the stack frame information when we create the instance via StackWalker.getInstance().

# Process API

---

- The Process API lacks some key functionality, so developers have to write a messy code to perform those tasks.
- For example, in order to do something as simple as getting your process PID in earlier versions of Java, you would need to either access native code or use some sort of a workaround.
- The process of doing so is very cumbersome.
- Java 9 comes to fix those issues and provide a clean API for interaction with processes.
- It enhances the Process class and introduces ProcessHandle with its nested interface Info to overcome the limitations we had in the past.
- More specifically two new interfaces have been added to the JDK:
  - 1. `java.lang.ProcessHandle`
  - 2. `java.lang.ProcessHandle.Info`

Modifier and Type	Method	Description
static Stream<ProcessHandle>	allProcesses()	It returns a snapshot of all processes visible to the current process.
Stream<ProcessHandle>	children()	It returns a snapshot of the current direct children of the process.
int	compareTo(ProcessHandle other)	It compares this ProcessHandle with the specified ProcessHandle for the order.
static ProcessHandle	current()	It returns a ProcessHandle for the current process.
Stream<ProcessHandle>	descendants()	It returns a snapshot of the descendants of the process.
boolean	destroy()	It requests the process to be killed.
boolean	destroyForcibly()	It requests the process to be killed forcibly.
boolean	equals(Object other)	It returns true if another object is non-null, is of the same implementation, and represents the same system process; otherwise, it returns false.
int	hashCode()	It returns a hash code value for this ProcessHandle.
ProcessHandle.Info	info()	It returns a snapshot of information about the process.

ProcessHandle		
int	hashCode()	It returns a hash code value for this ProcessHandle.
ProcessHandle.Info	info()	It returns a snapshot of information about the process.
boolean	isAlive()	It tests whether the process represented by this ProcessHandle is alive.
static Optional<ProcessHandle>	of(long pid)	It returns an Optional<ProcessHandle> for an existing native process.
CompletableFuture<ProcessHandle>	onExit()	It returns a CompletableFuture<ProcessHandle> for the termination of the process.
Optional<ProcessHandle>	parent()	It returns an Optional<ProcessHandle> for the parent process.
long	pid()	It returns the native process ID of the process.
boolean	supportsNormalTermination()	It returns true if the implementation of destroy() normally terminates the process.

## Http/2 Client

---

- HTTP/2 Client: HTTP/2 client is one of the feature of JDK 9.
- HTTP/2 is the newest version of the HTTP Protocol.
- By the help of HTTP/2 client, from the Java application, we can send the HTTP request and we can process the HTTP response.

# The Multi-Resolution API

---

- Java 9 introduced a new interface called `MultiResolutionImage` (JEP 251) to encapsulate a set of images with different resolutions into a single multi-resolution image.
- The APIs related to multi-resolution images are available under `java.awt.image` package and helps us to programmatically:
  - Get all variants of a particular image.
  - Get an image specific to the resolution required based on DPI metric and image transformations.
  - Let us learn more about the APIs for multi-resolution images introduced in Java 9.

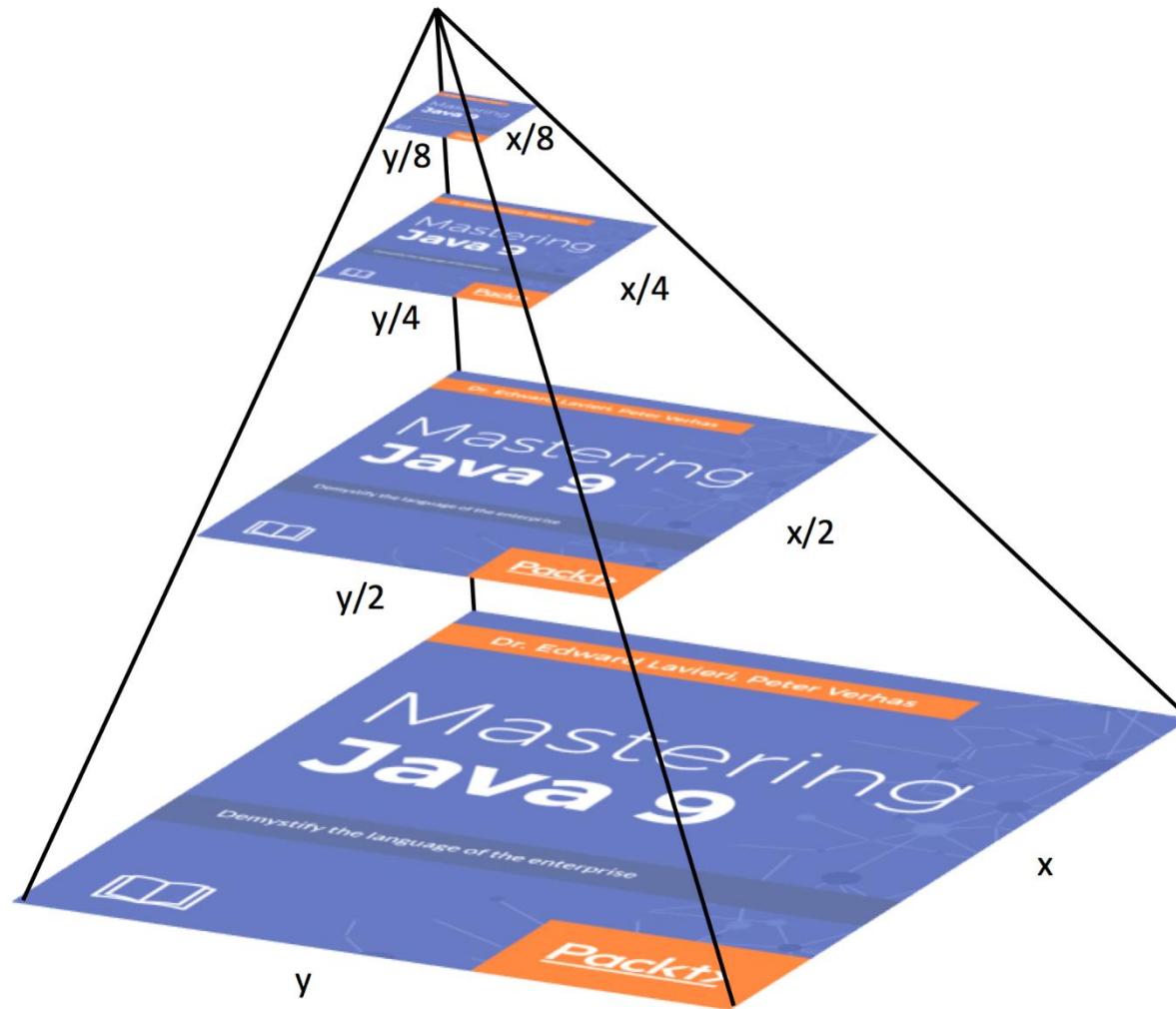
# MultiResolutionImage Interface

---

- There are two important functions in the MultiResolutionImage.
- `getResolutionVariant()` – This method returns an instance of `java.awt.Image` from the set of variants based on the width and height of the image given as parameters to the function.
- It will throw an `IllegalArgumentException` if the double values representing the dimensions are zero, negative, infinity or Not A Number.
- `getResolutionVariants()` – This method returns all the variants available for the given image.
- It returns a List containing objects of type `java.awt.Image`.

# MultiResolution Image

---



## What does the Java 9 JLink tool do

---

- The java 9 JLink tool enables you to create a runtime application by assembling a specific set of modules.
- With this tool, Java 9 now has the capability to create an application that works even if you don't have the JRE installed.
- JLink creates a minimal JRE required to run your application.
- It allows optimizing the runtime for the modules that you have included.

## What are the advantages of Java 9 JLink tool

---

- The JLink tool adds new functionalities and also opens up a few possibilities. Here are some of the advantages.
  - The runtime image can be optimized since it is possible to optimize the custom application module along with the platform modules.
  - Since the runtime created by jLink contains only the required modules, the size of the module is small. This opens up the possibility to use the application inside embedded and IOT devices.
  - The user does not have to install the JRE
  - Since the application image contains a subset of modules, it is safer from a security point of view since there would be fewer vulnerabilities

## What are the disadvantages of Java 9 JLink tool

---

- There are a few caveats to using the JLink tool.
  - The application image once created, cannot be updated or patched. For any changes, a new application needs to be deployed
  - When the user updates the JRE on her machine, the runtime image created by JLink does not get those updates and so any security fixes will not be applied.

# Jlink Tool

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19042.1110]
(c) Microsoft Corporation. All rights reserved.

F:\java9ws\java9moduledemo>tree /a /f
Folder PATH listing for volume New Volume
Volume serial number is 5641-E892
F:.
| .classpath
| .project
| pom.xml
|
+---.settings
|   org.eclipse.jdt.core.prefs
|   org.eclipse.m2e.core.prefs
|
+---src
|   +---main
|   |   +---java
|   |   |       module-info.java
|   |   |
|   |   \---com
```

# Jlink Tool

```
C:\Windows\System32\cmd.exe
+---src
|   +---main
|   |   +---java
|   |   |       module-info.java
|   |   \---com
|   |       \---boa
|   |           \---java9moduledemo
|   |               \---utilities
|   |                   LoggerTest.java
|
|   \---resources
\---test
    +---java
    \---resources
\---target
    +---classes
    |   module-info.class
    |
    +---com
    |   \---boa
```

The screenshot shows a Windows command prompt window titled "C:\Windows\System32\cmd.exe". The window displays a hierarchical file tree structure. The root directory is "src". Inside "src", there are "main", "resources", and "test" directories. The "main" directory contains a "java" folder which holds "module-info.java" and a "com" folder containing a "boa" folder with "java9moduledemo" and "utilities" subfolders, and a file named "LoggerTest.java". The "resources" and "test" directories also contain "java" and "resources" subfolders respectively. Below "src" is a "target" directory containing "classes" and "com" folders. The "classes" folder contains "module-info.class". The "com" folder contains a "boa" folder. The taskbar at the bottom of the screen shows various pinned icons for Microsoft Office applications like File Explorer, Edge, and Word, along with system status icons for battery level, signal strength, and weather.

# Jlink Tool

```
C:\Windows\System32\cmd.exe
+-- src
|   +-- main
|   |   +-- java
|   |   |   module-info.java
|   |   \-- com
|   |       +-- boa
|   |           +-- java9moduledemo
|   |               +-- utilities
|   |                   LoggerTest.java
|
|   \-- resources
\-- test
    +-- java
    \-- resources
\-- target
    +-- classes
    |   module-info.class
    |
    +-- com
        \-- boa
```

# Jlink Tool

```
C:\Windows\System32\cmd.exe

F:\java9ws\java9moduledemo>javac --release 9 -d mods/com.boa.java9moduledemo/utilities sr
c/main/java/module-info.java src/main/java/com/boa/java9moduledemo/utilities/LoggerTest.j
ava

F:\java9ws\java9moduledemo>
```

# Jlink Tool

```
C:\Windows\System32\cmd.exe
jdk.zipfs@11.0.12

F:\java9ws\java9moduledemo>jlink --add-modules java.base --output javabasert

F:\java9ws\java9moduledemo>
```

In fact, with the `--add-modules` option we add the `java.base` module to the runtime, while with the `--output` option we give the name to the runtime (`javabasert`).

# Jlink Tool

```
C:\Windows\System32\cmd.exe
jdk.zipfs@11.0.12

F:\java9ws\java9moduledemo>jlink --add-modules java.base --output javabasert

F:\java9ws\java9moduledemo>javabasert\bin\java --list-modules
java.base@11.0.12

F:\java9ws\java9moduledemo>jlink --add-modules java.base,java.desktop --output javabasedeskrt

F:\java9ws\java9moduledemo>javabasedeskrt\bin\java --list-modules
java.base@11.0.12
java.datatransfer@11.0.12
java.desktop@11.0.12
java.prefs@11.0.12
java.xml@11.0.12

F:\java9ws\java9moduledemo>
```

# Jlink Tool

D:\git\repository\java11jul2021\java9moduledemo>javabasertv3\bin\java -jar target\java9moduledemo-0.0.1-SNAPSHOT.jar  
Aug 04, 2021 12:23:40 PM com.boa.java9moduledemo.utilities.LoggerTest main  
INFO: Running test application..

D:\git\repository\java11jul2021\java9moduledemo>javabasertv3\bin\java --list-modules  
java.base@11.0.12  
java.datatransfer@11.0.12  
java.desktop@11.0.12  
java.logging@11.0.12  
java.prefs@11.0.12  
java.xml@11.0.12

The screenshot shows a video call interface with a "You're sharing your screen" overlay. On the left is a command prompt window showing Java module listing and running a test application. On the right is a "Chat" panel with a message history from Harish Kailasa and Sandesh Sadhale.

Chat
thank you from Harish Kailasa (privately): 12:00 PM
can you show the command once from Harish Kailasa (privately): 12:02 PM
yes from Sandesh Sadhale to everyone: 12:08 PM
Yeah from Harish Kailasa (privately): 12:08 PM
2 mins

D:\git\repository\java11jul2021\java9moduledemo>javabasertv3\bin\java -jar target\java9moduledemo-0.0.1-SNAPSHOT.jar  
Aug 04, 2021 12:23:40 PM com.boa.java9moduledemo.utilities.LoggerTest main  
INFO: Running test application..

D:\git\repository\java11jul2021\java9moduledemo>

The screenshot shows a video call interface with a "You're sharing your screen" overlay. On the left is a command prompt window showing Java module listing and running a test application. On the right is a "Chat" panel with a single message from Harish Kailasa.

Chat
thank you from Harish Kailasa (privately): 12:00 PM

## Java 10 Changes

---

- Docker Awareness and Support
- Unmodifiable Collections
- Garbage Collection Enhancements
- Application Class Data Sharing
- Ahead-of-Time Compilation

# Java 10 Changes

---

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

## Docker Awareness and Support

---

- Starting from Java 8 update 131, a number of features are introduced to Java to improve getting the correct resource limits when running in a Docker containers.
- JDK 10 now has docker awareness. So we no longer have to change JVM parameters to suit Container deployment
- Docker awareness. When running on Linux systems, the Java Virtual Machine (JVM) will know if it is running in a Docker container.
- Container-specific information—the number of CPUs and total memory allocated to the container—will be extracted by the JVM instead of it querying the operating system.

## Docker Awareness and Support

---

- With the release of Java 10, the JVM now recognizes constraints set by container control groups (cgroups).
- Both memory and cpu constraints can be used manage Java applications directly in containers, these include:
  - adhering to memory limits set in the container
  - setting available cpus in the container
  - setting cpu constraints in the container
  - Java 10 improvements are realized in both Docker for Mac or Windows and Docker Enterprise Edition environments

## Container Memory Limits

---

- Until Java 9 the JVM did not recognize memory or cpu limits set by the container using flags.
- In Java 10, memory limits are automatically recognized and enforced.
- Java defines a server class machine as having 2 CPUs and 2GB of memory and the default heap size is  $\frac{1}{4}$  of the physical memory.
- For example, a Docker Enterprise Edition installation has 2GB of memory and 4 CPUs. Compare the difference between containers running Java 8 and Java 10.

# Container Memory Limits

---

- First, Java 8:
- docker container run -it -m512 --entrypoint bash openjdk:latest
- \$ docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize    uintx MaxHeapSize := 524288000 {product} openjdk version "1.8.0\_162"

# Container Memory Limits

---

- The max heap size is 512M or  $\frac{1}{4}$  of the 2GB set by the Docker EE installation instead of the limit set on the container to 512M.
- In comparison, running the same commands on Java 10 shows that the memory limit set in the container is fairly close to the expected 128M.
- **docker container run -it -m512M --entrypoint bash openjdk:10-jdk**
- **In bash, run**
- **docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize**

# Container Memory Limits

```
Administrator: Command Prompt - docker container run -it -m512M --entrypoint bash openjdk:10-jdk
C:\WINDOWS\system32>docker container run -it -m512M --entrypoint bash openjdk:10-jdk
Unable to find image 'openjdk:10-jdk' locally
10-jdk: Pulling from library/openjdk
16e82e17faef: Pull complete
117dc02416a3: Pull complete
7e4c717259ac: Pull complete
7a518b8f48be: Pull complete
add32d44f708: Pull complete
a0158fa08543: Pull complete
9eb8cb7aab26: Pull complete
a9448aba0bc3: Pull complete
Digest: sha256:9f17c917630d5e95667840029487b6561b752f1be6a3c4a90c4716907c1aad65
Status: Downloaded newer image for openjdk:10-jdk
root@a3f0930dbc2c:/# docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize
    size_t MaxHeapSize                      = 134217728          {product} {ergonomic}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
root@a3f0930dbc2c:/# docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize
    size_t MaxHeapSize                      = 134217728          {product} {ergonomic}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
root@a3f0930dbc2c:/#
```



Type here to search



28°C

06:23 04/08/2021 ENG 20

## Setting Available CPUs

---

- By default, each container's access to the host machine's CPU cycles is unlimited.
- Various constraints can be set to limit a given container's access to the host machine's CPU cycles.
- Java 10 recognizes these limits:
- `docker container run -it --cpus 2 openjdk:10-jdk`
- `jshell> Runtime.getRuntime().availableProcessors()`

# Container Memory Limits

```
Administrator: Command Prompt - docker container run -it -m512M --entrypoint bash openjdk:10-jdk
C:\WINDOWS\system32>docker container run -it -m512M --entrypoint bash openjdk:10-jdk
Unable to find image 'openjdk:10-jdk' locally
10-jdk: Pulling from library/openjdk
16e82e17faef: Pull complete
117dc02416a3: Pull complete
7e4c717259ac: Pull complete
7a518b8f48be: Pull complete
add32d44f708: Pull complete
a0158fa08543: Pull complete
9eb8cb7aab26: Pull complete
a9448aba0bc3: Pull complete
Digest: sha256:9f17c917630d5e95667840029487b6561b752f1be6a3c4a90c4716907c1aad65
Status: Downloaded newer image for openjdk:10-jdk
root@a3f0930dbc2c:/# docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize
    size_t MaxHeapSize                      = 134217728          {product} {ergonomic}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
root@a3f0930dbc2c:/# docker-java-home/bin/java -XX:+PrintFlagsFinal -version | grep MaxHeapSize
    size_t MaxHeapSize                      = 134217728          {product} {ergonomic}
openjdk version "10.0.2" 2018-07-17
OpenJDK Runtime Environment (build 10.0.2+13-Debian-2)
OpenJDK 64-Bit Server VM (build 10.0.2+13-Debian-2, mixed mode)
root@a3f0930dbc2c:/#
```



Type here to search



28°C

06:23 04/08/2021 ENG 20

## Container CPU Limits

---

- All CPUs allocated to Docker EE get the same proportion of CPU cycles.
- The proportion can be modified by changing the container's CPU share weighting relative to the weighting of all other running containers.
- The proportion will only apply when CPU-intensive processes are running.
- When tasks in one container are idle, other containers can use the leftover CPU time.
- The actual amount of CPU time will vary depending on the number of containers running on the system. These can be set in Java 10:

## Container CPU Limits

---

- docker container run -it --cpu-shares 2048 openjdk:10-jdk
  - jshell> Runtime.getRuntime().availableProcessors()
  - \$1 ==> 2
- 
- The cpuset constraint sets which CPUs allow execution in Java 10.
  - docker run -it --cpuset-cpus="1,2,3" openjdk:10-jdk
  - jshell> Runtime.getRuntime().availableProcessors()
  - \$1 ==> 3

# Unmodifiable Collection

---

- "An unmodifiable collection is a collection, all of whose mutator methods ... are specified to throw `UnsupportedOperationException`.
- Such a collection thus cannot be modified by calling any methods on it.
- For a collection to be properly unmodifiable, any view collections derived from it must also be unmodifiable."
- Regarding modifications: "An unmodifiable collection is not necessarily immutable.
- If the contained elements are mutable, the entire collection is clearly mutable, even though it might be unmodifiable. ...
- However, if an unmodifiable collection contains all immutable elements, it can be considered effectively immutable."

# Unmodifiable Collection

---

- "An unmodifiable collection is a collection, all of whose mutator methods ... are specified to throw `UnsupportedOperationException`.
- Such a collection thus cannot be modified by calling any methods on it.
- For a collection to be properly unmodifiable, any view collections derived from it must also be unmodifiable."
- Regarding modifications: "An unmodifiable collection is not necessarily immutable.
- If the contained elements are mutable, the entire collection is clearly mutable, even though it might be unmodifiable. ...
- However, if an unmodifiable collection contains all immutable elements, it can be considered effectively immutable."

## Unmodifiable Collection

---

- In Java 10, following new static methods to create unmodifiable collections have been added.
- `java.util.List: <E> List<E> copyOf(Collection<? extends E> coll)`
- `java.util.Set: <E> Set<E> copyOf(Collection<? extends E> coll)`
- `java.util.Map: <K,V> Map<K,V> copyOf(Map<? extends K, ? extends V> coll)`
- Since the return collection/map is unmodifiable, attempting to modify it (adding/removing new elements) throws `UnsupportedOperationException`.

## Unmodifiable Collection

---

- `Collections.unmodifiableList()` (and other similar methods) returns a unmodifiable view of the source collection, so changes made to the source collection reflects in it.
- Whereas, in case of `List.copyOf()` method, if the source collection is subsequently modified, the returned List will not reflect such modifications..

## Difference between `copyOf` and `Collections.unmodifiableList`

---

- The `Collections.unmodifiableList` method returns an unmodifiable view of the source List.
- So if the source List is modified, these changes are reflected in the unmodifiable List.
- The `copyOf` methods on the other hand, return a read-only copy of the source Collection.
- So even if the source Collection is modified, the unmodifiable Collection does not change.

## GC Options

---

- `java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar java-application.jar`
- `javabasertv3\bin\java -Xmx12m -Xms3m -Xmn1m -XX:+UseG1GC -XX:+PrintGCDetails -jar target/java9moduledemo-0.0.1-SNAPSHOT.jar`

## Ahead Of Time vs Just In Time in Java

---

- Byte Code To Machine Code:
- Byte code is get converted to machine level code using a dictionary of instructions (byte to m/c), because of different types of machines (like Ubuntu, mac, windows, etc) have different types of the instruction set.
- The interpreter is very quick to start and load the app.
- One problem is interpreter does not perform any kind of optimization, because of that same byte code get converted to machine code every time.

# LOCAL-VARIABLE TYPE INFERENCE

---

- Local variable
  - Only local variables are supported
  - No methods parameters
  - No return types
  - No fields
  - No lambdas

## LOCAL-VARIABLE TYPE INFERENCE

---

- This feature is all about the variables in Java, whenever we declare a variable on the left-hand side we define a data type with the name of the variable and on the right-hand side, we define the expression to initialize the variable.
- Since we are talking about local variables here, these variables are limited to method scope i.e we are talking about variables inside the methods.

## LOCAL-VARIABLE TYPE INFERENCE

---

- Local variable type inference- The type for the local variables i.e variables declared inside the method will be inferred automatically by the compiler.
- We do not have to tell the compiler about the type for a variable, it will be inferred by the compiler itself.

# LOCAL-VARIABLE TYPE INFERENCE

---

- For example
  - Before Java 10 – String name = “Deepak”;
  - After Java10 – var name = “Deepak”;
- 
- public void printName() {
  - var name = “Deepak”;
  - System.out.println(“My name is ” + name);
  - }

# LOCAL-VARIABLE TYPE INFERENCE

---

- Why do we need Local variable type inference?
  - 1. Type less code.
  - 2. Makes things simple.
  - 3. Clean code.

# LOCAL-VARIABLE TYPE INFERENCE

---

- // Before var
  - URL url = new URL("http://javamodularity.com");
  - URLConnection connection = url.openConnection();
  - BufferedInputStream inputStream =
  - new BufferedInputStream(connection.getInputStream());
- 
- // After var
  - var bookurl = new URL("http://javamodularity.com");
  - var connection = bookurl.openConnection();
  - var bookStream = new  
  BufferedInputStream(connection.getInputStream());

# LOCAL-VARIABLE TYPE INFERENCE

---

- When it can go wrong?
- You cannot just overuse or abuse var just for the sake of simplicity and reducing the amount of code you write, consider the code below to understand it better.
- `var result = aService.findTheThing();`
- It is said that the code is written only once and read many times.
- In the example above, it won't be wise to use var as first of all, the method is not named very well.
- Also, it will be difficult here to determine the type of variable result.
- So, instead of just using the var here, we can probably give it type if can really not fix the method name.
- This code will be read by many developers during code reviews and others, so it is better to not use vars here.

# LOCAL-VARIABLE TYPE INFERENCE

---

- Some facts about var
- Var is not a keyword – Var is not a keyword, as introducing a new keyword in Java is really a big change.
- Also, keywords in Java cannot be used as identifiers. Var is a reserved type, not a keyword.
- So, it is okay to use the following statement.
- `var var = "var";`

# LOCAL-VARIABLE TYPE INFERENCE

---

- Java turned into a dynamically typed language – No, even with this change we cannot say that Java turned into dynamically typed language as the compiler will just infer the type using the right-hand side expression.
- However, internally it will actually assign the type.
- Let's consider an example, where we will use a piece of code with and without var and see the compiled version to check if there is a difference between these two.
- // Without var
- int counter = 0;
- Counter = counter + 1;
- // With var
- var counter = 0;
- counter = counter + 1;

# LOCAL-VARIABLE TYPE INFERENCE

---

- Is Type inference new in Java?
  - The answer is no, the type inference is not new, we have seen this before as well
- Before java 7, we used to provide types explicitly
  - `List list = new ArrayList();`
- Java7 onwards
  - `List list = new ArrayList<>();`
  - The only difference is, type inference will happen from left to right.

# LOCAL-VARIABLE TYPE INFERENCE

---

- Limitations of Type Inference
  - 1. Lambdas must have an explicit target type – As there will be multiple functional interfaces that take input A and return B and in this case the compiler will get confused and will not be able to determine the type.
  - 2. Var in combination with diamond operator – Consider the example below
  - Var list = new ArrayList<>();
  - In this example, the statement will compile. However, the type would be Object. To continue using the var with a specific type, use the type with a diamond operator like below.
  - Var list = new ArrayList();
  - 3. Local type inference only – Only applicable to local variables inside the methods.

# LOCAL-VARIABLE TYPE INFERENCE

---

- Limitations of Type Inference
  - 4. Var won't work without initializer – If you are using var without initializing the variable, it won't work and give you errors.
  - Var name; // won't work
  - Var name = “Deepak”; // works fine
  - 5. Var won't work with array initializer – If you are using vars with array initializer like below, it won't work
  - Var array = {1, 2, 3, 4}; // would produce error.

## Java 11 Features

---

- Type inference for the arguments of Lambda expressions
- Simplified launch of single file programs
- Evolution of the http client api

# Java 11 Features

---

- `java.lang.String`:
  - boolean `isBlank ()`: Returns true if the String is empty or composed only of whitespaces, otherwise false.
  - `Stream<String> lines ()`: Returns a stream of lines extracted from the String.
  - `String repeat (int)`: returns a String that is the original String concatenated with itself n times.
  - `String strip ()`: returns a String free of trailing and leading whitespaces. To put it simply, `strip ()` is the “Unicode-aware” version of `trim ()` whose definition of whitespace dates back from the first versions of Java.
  - `String stripLeading ()`: Returns a String free of its leading whitespaces.
  - `String stripTrailing ()`: returns a String free of trailing whitespaces.

# Java 11 Features

---

- `java.util.function.Predicate`
  - `Predicate not (Predicate)`. Returns a `Predicate` which is the negation of the `Predicate` passed as an argument. Example to filter a list:
- `java.lang(CharSequence`
  - `int compare (CharSequence, CharSequence)`: compares two instances of `CharSequence` in lexicographic order. Returns a negative, null or positive value.
- `java.lang.StringBuffer / java.lang.StringBuilder`
  - Both classes now have access to a new `compareTo ()` method that takes a `StringBuffer / StringBuilder` argument and returns an `int`. The comparison logic follows the same lexicographic order as for the new method of the `CharSequence` class.

## Java 11 Features

---

- Visibility management of nested classes attributes
- Dynamic class-file constants
- The epsilon garbage collector
- Unicode 10

## Java 11 Features

---

- Deleted modules
- The Java EE and CORBA modules, after being deprecated in JAVA SE 9, were removed from JAVA SE Platform and JDK 11. The impacted modules are:
  - corba
  - transaction
  - activation
  - xml.bind
  - xml.ws
  - xml.ws.annotation

# Java 11 Features

- **JEP 323: Local-Variable Syntax for Lambda Parameters**
- This JEP adds support for the var keyword in lambda parameters.

```
List<String> list = Arrays.asList("a", "b", "c");
String result = list.stream()
    .map((var x) -> x.toUpperCase())
    .collect(Collectors.joining(","));
System.out.println(result2);
```

```
import org.jetbrains.annotations.NotNull;

List<String> list = Arrays.asList("a", "b", "c", null);
String result = list.stream()
    .map(@NotNull var x) -> x.toUpperCase()
    .collect(Collectors.joining(","));
System.out.println(result3);
```

## Java 11 Features

---

- **JEP 330: Launch Single-File Source-Code Programs**
- This Single-File Source-Code Program means the entire Java program in a single source code .java file. This JEP is a friendly feature in the early stage of learning Java, but not much benefit in Java development, we all use IDE.

# Java 11 Features

---

```
HelloJava.java
```

```
public class HelloJava {  
  
    public static void main(String[] args) {  
  
        System.out.println("Hello World!");  
  
    }  
}
```

# Java 11 Features

---

Before Java 11.

```
Terminal  
  
$ javac HelloJava.java  
  
$ java HelloJava  
  
Hello World!
```

Now Java 11.

```
Terminal  
  
$ java HelloJava.java  
  
Hello World!
```

# Java 11 Features



```
C:\Windows\System32\cmd.exe
    7 Dir(s) 79,228,366,848 bytes free

G:\Local disk\Java9>java Test.java

G:\Local disk\Java9>java Test.java
Hi.....
G:\Local disk\Java9>
```

# Http Client Enhancement

---

- Java had HttpURLConnection class for long time for HTTP communication.
- But over the time, requirements have gone complex and more demanding in applications.
- Before Java 11, developers had to resort to feature-rich libraries like Apache HttpComponents or OkHttp etc.
- We saw Java 9 release to include an HttpClient implementation as an experimental feature.
- It has grown over time and now a final feature of Java 11.
- Now the Java applications can make HTTP communications without the need to any external dependency.

## How to use HttpClient

---

- A typical HTTP interaction with the `java.net.http` module looks like-
- Create an instance of `HttpClient` and configure it as needed.
- Create an instance of `HttpRequest` and populate the information.
- Pass the request to the client, perform the request, and retrieve an instance of `HttpResponse`.
- Process the information contained in the `HttpResponse`.
- HTTP API can handle synchronous and asynchronous communication.

## Http Client Enhancement

---

- The major change in Java 11 was the standardization of HTTP client API that implements HTTP/2 and Web Socket.
- It aims to replace the legacy HttpURLConnection class which has been present in the JDK since the very early years of Java.

## Http Client Enhancement

---

- The incubated HTTP API from Java 9 is now officially incorporated into the Java SE API. The new HTTP APIs can be found in `java.net.HTTP`.\*
- The newer version of the HTTP protocol is designed to improve the overall performance of sending requests by a client and receiving responses from the server.
- This is achieved by introducing a number of changes such as stream multiplexing, header compression and push promises.
- As of Java 11, the API is now fully asynchronous (the previous HTTP/1.1 implementation was blocking).
- Asynchronous calls are implemented using `CompletableFuture`.

# Http Client Enhancement

---

- The CompletableFuture implementation takes care of applying each stage once the previous one has finished, so this whole flow is asynchronous.
- The new HTTP client API provides a standard way to perform HTTP network operations with support for modern Web features such as HTTP/2, without the need to add third-party dependencies.
- The new APIs provide native support for HTTP 1.1/2 WebSocket. The core classes and interface providing the core functionality include:
  - The HttpClient class, `java.net.http.HttpClient`
  - The HttpRequest class, `java.net.http.HttpRequest`
  - The `HttpResponse<T>` interface, `java.net.http.HttpResponse`
  - The WebSocket interface, `java.net.http.WebSocket`

# HTTP/2 Client

---

- The new HTTP client API provides the following:



A simple and concise API to deal with most HTTP requests

Support for HTTP/2 specification

Better performance

Better security

A few more enhancements

# Improved Java Doc – HTML5 compatibility and Search

The screenshot shows a Java API documentation page. At the top, there's a navigation bar with links for OVERVIEW, MODULE, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. To the right of the navigation bar, it says "Java™ Platform Standard Ed. 9 DRAFT 9-ea+162-jigsaw-nightly-h6252-20170328". Below the navigation bar, there are links for PREV, NEXT, FRAMES, NO FRAMES, and ALL CLASSES. On the far right, there's a search bar with a magnifying glass icon and the word "Map", and an "X" button.

**Types**

- java.util.Map
- java.nio.channels.FileChannel.MapMode
- java.nio.MappedByteBuffer
- java.util.AbstractMap
- javax.swing.ActionMap
- javax.swing.plaf.ActionMapUIResource
- javax.activation.CommandMap
- javax.swing.ComponentInputMap
- javax.swing.plaf.ComponentInputMapUIResource
- java.util.concurrent.ConcurrentHashMap
- java.util.concurrent.ConcurrentMap
- java.util.concurrent.ConcurrentNavigableMap
- java.util.concurrent.ConcurrentSkipListMap

**Modules**

Module	Description
java.activation	Defines the Java Activation Framework
java.base	Defines the foundation of the Java SE platform
java.compiler	Defines the Java Compiler API
java.desktop	Defines the Java Desktop API

## The Java versioning scheme

---

- Java version-string is a format that contains version specific information.
- This version-string consists of major, minor, security and patch update releases.
- In Java 9, a new version-string scheme is introduced that looks like the below.
- \$MAJOR.\$MINOR.\$SECURITY.\$PATCH

# The Java versioning scheme

---

- \$MAJOR
  - This version number shows a major change in Java version.
  - It increases when a major change occurs.
  - Like Java 8 to Java 9. Each major release contains new features to the existing one.
- \$MINOR
  - This version number shows minor changes in Java version and increases with each minor update.
  - These updates can be bug fixes, revision to standard API etc.
  - if an update is released to Java 9, version-string format will be Java 9.1 (contains major and minor release number).

# The Java versioning scheme

---

- A version number, \$VNUM, is a non-empty sequence of elements separated by period characters (U+002E).
- An element is either zero, or an unsigned integer numeral without leading zeros.
- The final element in a version number must not be zero.
- When an element is incremented, all subsequent elements are removed. The format is:
  - [1-9][0-9]\*((\.\.0)\*\.[1-9][0-9]\*)\*
- The sequence may be of arbitrary length but the first four elements are assigned specific meanings, as follows:
- \$FEATURE.\$INTERIM.\$UPDATE.\$PATCH

# Java Runtime.Version Class Signature

---

- **\$FEATURE** — The feature-release counter, incremented for every feature release regardless of release content.
  - Features may be added in a feature release; they may also be removed, if advance notice was given at least one feature release ahead of time.
  - Incompatible changes may be made when justified.
- **\$INTERIM** — The interim-release counter, incremented for non-feature releases that contain compatible bug fixes and enhancements but no incompatible changes, no feature removals, and no changes to standard APIs.
- **\$UPDATE** — The update-release counter, incremented for compatible update releases that fix security issues, regressions, and bugs in newer features.
- **\$PATCH** — The emergency patch-release counter, incremented only when it's necessary to produce an emergency release to fix a critical issue.

# Version strings

---

- A version string, `$VSTR`, is a version number `$VNUM`, as described above, optionally followed by pre-release and build information, in one of the following formats:
  - `$VNUM(-$PRE)?\+$BUILD(-$OPT)?`
  - `$VNUM-$PRE(-$OPT)?`
  - `$VNUM(+-$OPT)?`
- where:
  - `$PRE`, matching `([a-zA-Z0-9]+)` — A pre-release identifier. Typically `ea`, for a potentially unstable early-access release under active development, or `internal`, for an internal developer build.
  - `$BUILD`, matching `(0|[1-9][0-9]*)` — The build number, incremented for each promoted build. `$BUILD` is reset to 1 when any portion of `$VNUM` is incremented.
  - `$OPT`, matching `([-a-zA-Z0-9.]+)` — Additional build information, if desired. In the case of an internal build this will often contain the date and time of the build.

## Using Regular expression patterns with Predicate

---

- Compile regular expression into `java.util.function.Predicate`.
- This can be useful when you want to perform some operation on matched tokens.
- I have list of emails with different domain and I want to perform some operation only on email ids with domain name “example.com”.
- Now use `Pattern.compile().asPredicate()` method to get a predicate from compiled regular expression.
- This predicate can be used with lambda streams to apply it on each token into stream.

## Using Regular expression patterns with Predicate

---

- We can easily turn a Pattern into a Predicate with the asPredicate and asMatchPredicate methods.
- The asPredicate method return a predicate for testing if the pattern can be found given string.
- And the asMatchPredicate return a predicate for testing if the pattern matches a given string.

## Collection ToArray

---

- Object[] toArray()
  - Returns an array containing all of the elements in this collection.
  - If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.
  - The returned array's runtime component type is Object.
  - The returned array will be "safe" in that no references to it are maintained by this collection.
  - (In other words, this method must allocate a new array even if this collection is backed by an array). The caller is thus free to modify the returned array.

## Unicode 10 Standard

---

- Java 11 has upgraded existing platform APIs to support version 10.0 of the Unicode Standard.
- It supports the latest Unicode version, particularly in the classes below:
  - Character and String in the `java.lang` package
  - NumericShaper in the `java.awt.font` package
  - Bidi, BreakIterator, and Normalizer in the `java.text` package

## Unicode 10 Standard

---

- Java 11 will support Unicode 10. Since Java 10 supports Unicode 8, that means both Unicode 9 and Unicode 10 changes will be implemented.
- However, Unicode Collation Algorithm, Unicode Security Mechanisms, Unicode IDNA Compatibility Processing, and Unicode Emoji will not be implemented.

# USING STRINGS IN JAVA 11

---

- **String.repeat(Integer)**
  - This method simply repeats a string n times. It returns a string whose value is the concatenation of given string repeated N times.
  - If this string is empty or count is zero then the empty string is returned.

# USING STRINGS IN JAVA 11

---

- **String.isBlank()**
  - This method indicates whether a string is empty or contains only white-spaces. Previously, we have been using it from Apache's StringUtils.java.

# USING STRINGS IN JAVA 11

---

- **String.strip()**
  - This method takes care of removing leading and trailing white-spaces.
  - We can be even more specific by removing just the leading characters by using String.stripLeading() or just the trailing characters by using String.stripTrailing().

# USING STRINGS IN JAVA 11

---

- **String.lines()**
  - This method helps in processing multi-line texts as a Stream.

# JAVA 11: REMOVED FEATURES AND OPTIONS

---

- Removal of com.sun.awt.AWTUtilities Class
- Removal of Lucida Fonts from Oracle JDK
- Removal of appletviewer Launcher
- Oracle JDK's javax.imageio JPEG Plugin No Longer Supports Images with alpha
- Removal of sun.misc.Unsafe.defineClass
- Removal of Thread.destroy() and Thread.stop(Throwable) Methods
- Removal of sun.nio.ch.disableSystemWideOverlappingFileLockCheck Property
- Removal of sun.locale.formatasdefault Property
- Removal of JVM-MANAGEMENT-MIB.mib
- Removal of SNMP Agent
- Removal of Java Deployment Technologies
- Removal of JMC from the Oracle JDK
- Removal of JavaFX from the Oracle JDK
- JEP 320 Remove the Java EE and CORBA Modules

# JAVA 11: Deprecated Features and Options

---

- ThreadPoolExecutor Should Not Specify a Dependency on Finalization
- JEP 335 Deprecate the Nashorn JavaScript Engine
- Deprecate -XX+AggressiveOpts
- Obsolete Support for Commercial Features
- Deprecate Stream-Based GSSContext Methods
- JEP 336 Deprecate the Pack200 Tools and API

## Java 12 Features

---

- JVM Changes – JEP 189, JEP 346, JEP 344, and JEP 230.
- Switch Expressions
- File mismatch() Method
- Compact Number Formatting
- Teeing Collectors in Stream API
- Java Strings New Methods – indent(), transform(), describeConstable(), and resolveConstantDesc().
- JEP 334: JVM Constants API
- JEP 305: Pattern Matching for instanceof
- Raw String Literals is Removed From JDK 12.

# Switch Expression

```
String monthName = switch (monthNumber) {  
    case 1 -> {  
        System.out.println("January");  
        // yield statement is used in a Switch Expression  
        // break,continue statements are used in a Switch Statement  
        yield "January"; // yield mandatory!  
    }  
    case 2 -> "February";  
    case 3 -> "March";  
    case 4 -> "April";  
    default -> "Invalid Month";  
};
```

- Create expressions using switch statement
- Released in JDK 14
  - Preview - JDK 12 and 13

# Switch Expression

---

```
public static String findDayOfTheWeekWithSwitchExpression(int day) {  
    String dayOfWeek = switch(day) {  
        case 0 -> {  
            System.out.println("Do Some complex logic here");  
            yield "Sunday";  
        }  
        case 1 -> "Monday";  
        case 2 -> "Tuesday";  
        case 3 -> "Wednesday";  
        default -> throw new IllegalArgumentException("Invalid Option"+ day);  
    };  
  
    return dayOfWeek;  
}
```

## Java 12 String Enhancements

---

- `String text = "Hello RPS!\nThis is Java 12 article.;"`
- `text = text.indent(4);`
- `System.out.println(text);`
  
- `text = text.indent(-10);`
- `System.out.println(text);`

## File::mismatch Method

---

- Refer code

## String constants

---

- In JDK 12, two new interfaces, `java.lang.constant.Constable` and `java.lang.constant.ConstantDesc` have been implemented inside the `String` class.
- Each of these interfaces from Constants API has a method declared inside it with `String` class providing implementation for both of them.

## Java 14 Features

---

- Pattern Matching for instanceof (Preview)
- Packaging Tool (Incubator)
- NUMA-Aware Memory Allocation for G1
- JFR Event Streaming
- Non-Volatile Mapped Byte Buffers
- Helpful NullPointerExceptions
- Records (Preview)
- Switch Expressions (Standard)
- Deprecate the Solaris and SPARC Ports

## Java 14 Features

---

- Remove the Concurrent Mark Sweep (CMS) Garbage Collector
- ZGC on macOS
- ZGC on Windows
- Deprecate the ParallelScavenge + SerialOld GC Combination
- Remove the Pack200 Tools and API
- Text Blocks (Second Preview)
- Foreign-Memory Access API (Incubator)

# Text Block

---

```
System.out.println("First Line\nSecond Line\nThird Line");
System.out.println("""
    First Line
    Second Line
    Third Line""")
;
```

- Simplify Complex Text Strings
- Released in JDK 15
  - Preview - JDK 13 and 14

# Text Block

---

```
2
3 public class TextBlocksRunner {
4
5     public static void main(String[] args) {
6         String str = """
7             Line 1
8             Line 2
9             Line 3
10            """;
11         System.out.print(str);
12
13     }
14
15 }
16
```

# Records

---

```
record Person(String name, String email, String phoneNumber) { }
```

- Eliminate verbosity in creating Java Beans
  - Public accessor methods, constructor, equals, hashCode and toString are automatically created
  - You can create custom implementations if you would want
- Released in JDK 16
  - Preview - JDK 14 and 15
- Remember:
  - Compact Constructors are only allowed in Records
  - You can add static fields, static initializers, and static methods

# Records

---

```
public class RecordsRunner {  
  
    record Person(String name, String email, String phoneNumber) {}  
  
    public static void main(String[] args) {  
        Person person = new Person("Ranga", "ranga@in28minutes.com", "123-456-7890");  
        System.out.println(person);  
    }  
}
```

# Records

---

```
public class RecordsRunner {  
  
    record Person(String name, String email, String phoneNumber) {}  
  
    public static void main(String[] args) {  
        Person person = new Person("Ranga", "ranga@in28minutes.com", "123-456-7890");  
        Person person1 = new Person("Ranga", "ranga@in28minutes.com", "123-456-7890");  
        Person person2 = new Person("Ranga1", "ranga@in28minutes.com", "123-456-7890");  
        System.out.println(person.equals(person1));  
        System.out.println(person.equals(person2));  
        System.out.println(person.name());  
    }  
}
```

# Records

---

```
public class RecordsRunner {  
    record Person(String name, String email, String phoneNumber) {  
        Person(String name, String email, String phoneNumber) {  
            if(name == null)  
                throw new IllegalArgumentException("name is null");  
            this.name = name;  
            this.email = email;  
            this.phoneNumber = phoneNumber;  
        }  
    }  
    public static void main(String[] args) {  
        Person person = new Person("Ranga", "ranga@in28minutes.com", "123-456-7890");  
        System.out.println(person.name());  
    }  
}
```

# Records

---

```
public class RecordsRunner {  
    record Person(String name, String email, String phoneNumber) {  
        Person {  
            if(name == null)  
                throw new IllegalArgumentException("name is null");  
        }  
    }  
  
    public static void main(String[] args) {  
        Person person = new Person("Ranga", "ranga@in28minutes.com", "123-456-7890");  
        System.out.println(person.name());  
  
        // Person person1 = new Person("Ranga", "ranga@in28minutes.com", "123-456-7890");  
        // Person person2 = new Person("Ranga1", "ranga@in28minutes.com", "123-456-7890");  
        // System.out.println(person.equals(person1));  
        // System.out.println(person.equals(person2));
```

# Records

---

```
public class RecordsRunner {  
  
    record Person(String name, String email, String phoneNumber) {  
        Person {  
            if(name == null)  
                throw new IllegalArgumentException("name is null");  
        }  
  
        public String name() {  
            System.out.println("Do Something");  
            return name;  
        }  
  
    }  
  
    public static void main(String[] args) {  
        Person person = new Person("Ranga", "ranga@in28minutes.com", "123-456-7890");  
        System.out.println(person.name());  
    }  
}
```

# JShell

<https://cr.openjdk.java.net/~rfield/tutorial/JShellTutorial.html>



# JShell

```
Administrator: Command Prompt - jshell
Microsoft Windows [Version 10.0.19042.1110]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>java -version
java version "11.0.12" 2021-07-20 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.12+8-LTS-237)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.12+8-LTS-237, mixed mode)

C:\WINDOWS\system32>jshell
| Welcome to JShell -- Version 11.0.12
| For an introduction type: /help intro

jshell> System.out.println("Rocking....");
Rocking....  
  
jshell>  
  
jshell> 56757+466497-54767
$2 ==> 468487  
  
jshell> /list  
  
1 : System.out.println("Rocking....");
2 : 56757+466497-54767  
  
jshell>
```



# Jshell -Method

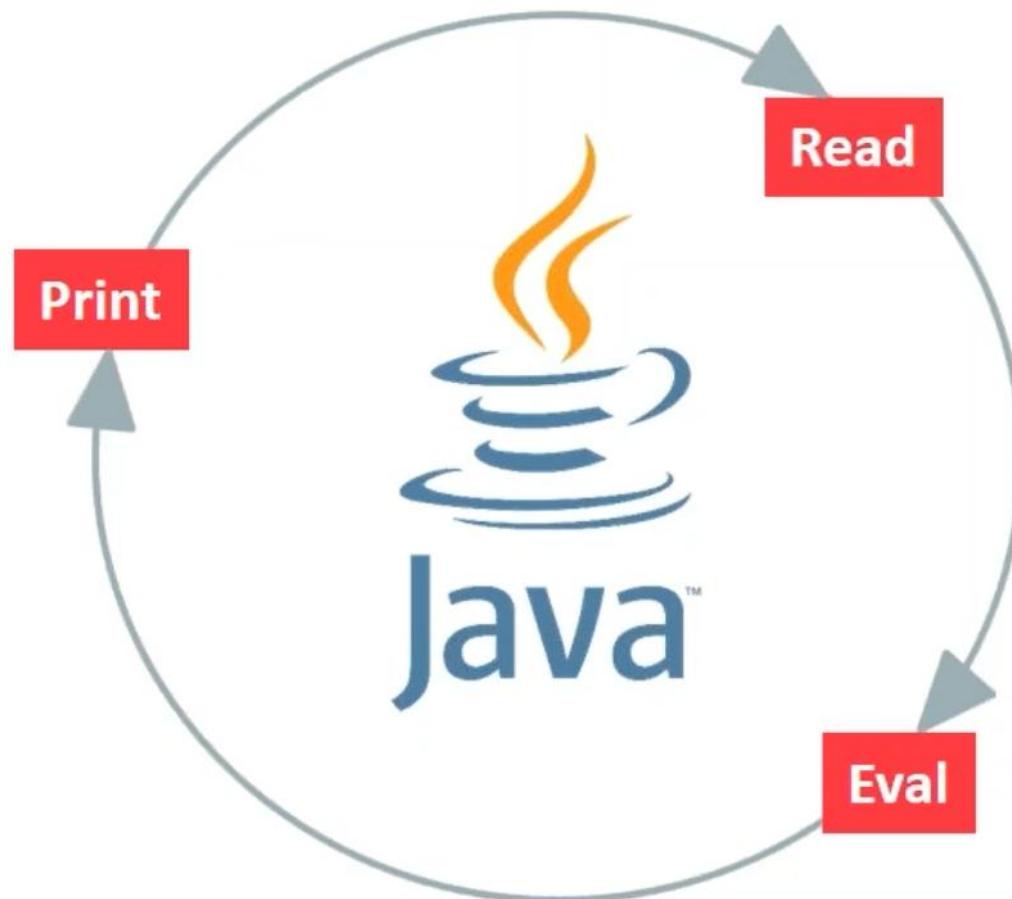
```
C:\Administrator: Command Prompt
C:\WINDOWS\system32>jshell
| Welcome to JShell -- Version 11.0.12
| For an introduction type: /help intro

jshell> void display(){
...>     System.out.println("Rocking....");
...> }
| created method display()

jshell> display();
Rocking....
```

# Repl-Jshell

---



# Collection Factory Methods

---



# Collection Factory Methods

```
C:\WINDOWS\system32>jshell
| Welcome to JSShell -- Version 11.0.12
| For an introduction type: /help intro

jshell> Set namesSet=Set.of("Param","Bala","Vignesh");
namesSet ==> [Param, Vignesh, Bala]

jshell> List dataList=List.of("Param",51,"1970-12-02", "Trainer");
dataList ==> [Param, 51, 1970-12-02, Trainer]

jshell>
```

```
C:\WINDOWS\system32>jshell
| Welcome to JSShell -- Version 11.0.12
| For an introduction type: /help intro

jshell> Map<Long,String> customerMap=Map.of(1L,"Bala",2L,"Param",3L,"Vignesh");
customerMap ==> {3=Vignesh, 1=Bala, 2=Param}

jshell> ■
```

# Jshell Import

In a JShell session, you get the following set of imports by default:

```
1 jshell> /imports
2 jshell> /imports
3 |   import java.io.*
4 |   import java.math.*
5 |   import java.net.*
6 |   import java.nio.file.*
7 |   import java.util.*
8 |   import java.util.concurrent.*
9 |   import java.util.function.*
10 |  import java.util.prefs.*
11 |  import java.util.regex.*
12 |  import java.util.stream.*
```

java

You can import other packages, but how does JShell know where to look for custom packages?

Consider the following case:

```
1 jshell> import com.mycompany.*
```

java

# Stream Enhancements

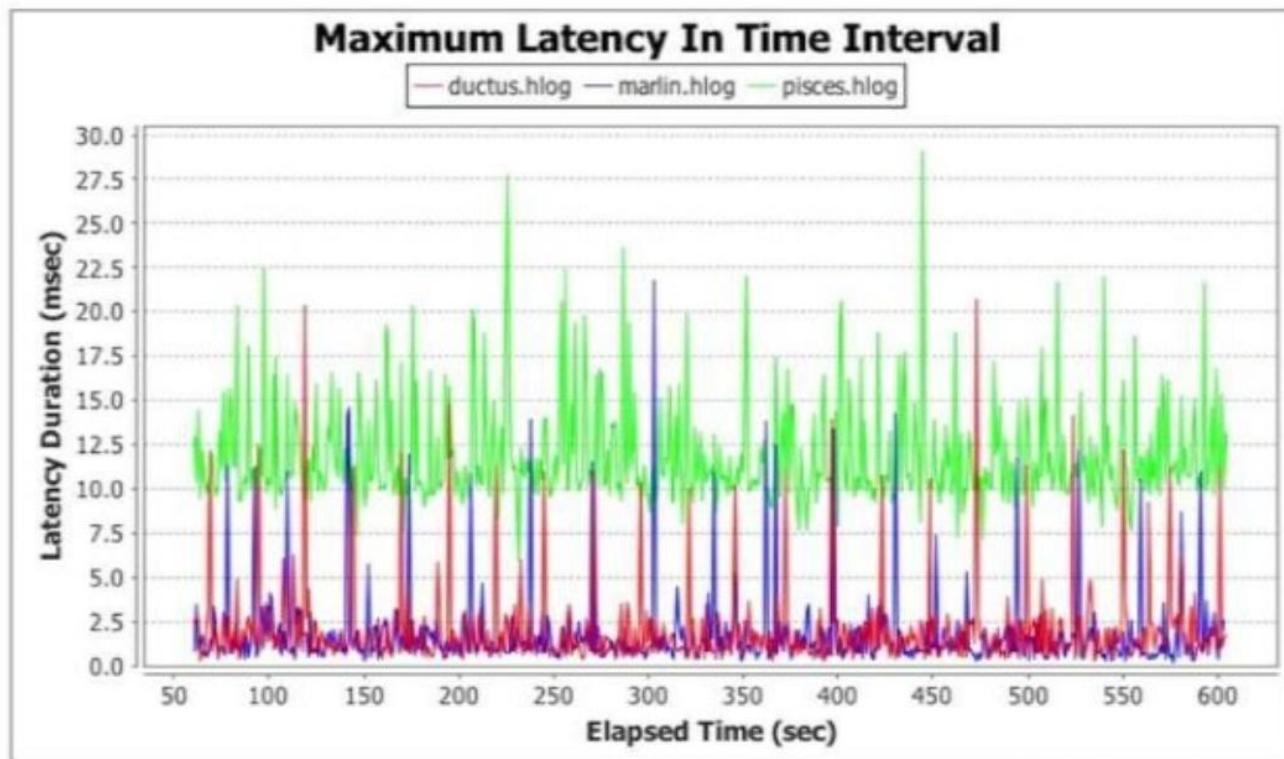
---

- **dropWhile()/takeWhile()**
  - Like skip/limit but uses Predicate rather than number
- **Improved iterate**
  - Enables a stream to be more like a for loop
- **Parallel Files.lines()**
  - Memory mapped, divided on line break boundary
- **Stream from Optional**
  - Stream of zero or one elements



# Marlin Graphics Renderer

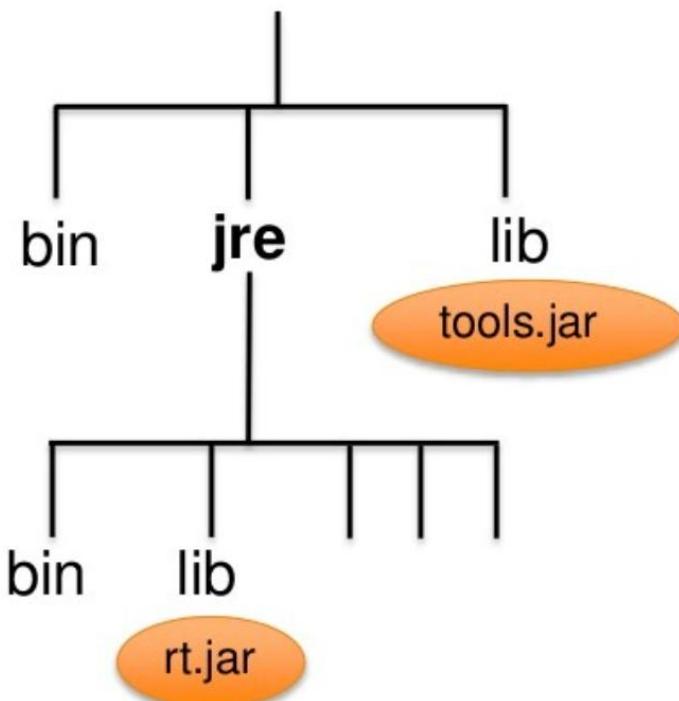
- Replaces Pisces open-source renderer
  - Comparable performance to closed-source Ductus



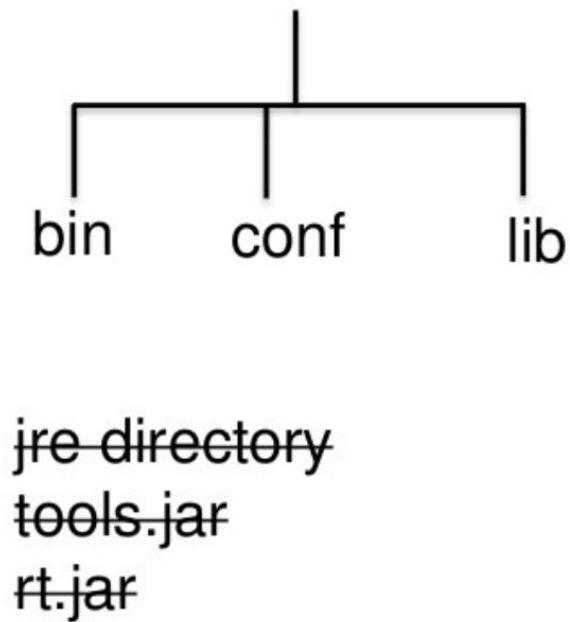
# JDK/JRE File Structure (JEP 220)

---

Pre-JDK 9



JDK 9



## Removed From JDK 9

---

- Six deprecated APIs (JEP 162)
  - ActionListeners in Pack200 and LogManager
- JRE version selection command line option (JEP 231)
  - -version: no longer accepted
- Demos and samples (JEP 298)
  - Out-of-date, unmaintained

## Removed From JDK 9

---

- **JVM TI hprof agent (JEP 240)**
  - Only ever intended as a demo of JVM TI
  - Useful features now in other tools (like jmap)
- **Remove the jhat tool (JEP 241)**
  - Experimental tool added in JDK 6
  - Unsupported
  - Better heap visualisation tools available

# Differences between Oracle JDK and OpenJDK

---

- Only Oracle JDK offers Solaris, only OpenJDK offers Alpine Linux.
- Oracle JDK offers "installers" (msi, rpm, deb, etc.) which not only place the JDK binaries in your system but also contain update rules.
- In some cases, handle some common configurations like set common environmental variables (such as, JAVA\_HOME in Windows) .
- It establishes file associations (such as, use java to launch .jar files). OpenJDK is offered only as compressed archive (tar.gz or .zip).
- javac —release for release values 9 and 10 behave differently. Oracle JDK binaries include APIs that were not added to OpenJDK binaries like javafx, resource management, and (pre JDK 11 changes) JFR APIs.
- Oracle JDK offers "JDK" and "JRE". OpenJDK offers only "JDK".

## Differences between Oracle JDK and OpenJDK

---

- Usage Logging is only available in Oracle JDK.
- OpenJDK will (continue to) throw an error.
- It halts if the `-XX:+UnlockCommercialFeatures` flag is used.
- Oracle JDK no longer requires the flag and will print a warning but continue execution if used.
- Oracle JDK requires that third party cryptographic providers be signed with an Oracle-provided certificate.
- OpenJDK will continue allowing the use of unsigned third-party crypto providers.

# Differences between Oracle JDK and OpenJDK

---

- The output of java -version will be different.
- Oracle JDK will say java and include LTS.
- OpenJDK (when produced by Oracle) will say OpenJDK and not include the Oracle-specific LTS identifier.
- Oracle JDK will be released under OTN License.
- Any License file will need to point to OTN.
- OpenJDK will be released under GPLv2wCP and will include the GPL license.
- Oracle JDK will distribute FreeType under the FreeType license and OpenJDK will do so under GPLv2.
- The contents of \legal\java.desktop\freetype.md will therefore be different.
- Oracle JDK has Java cup and steam icons and OpenJDK has Duke icons.
- Oracle JDK source code includes "ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms." and OpenJDK source includes the GPL.

# Types of GCs in Java

---

Serial Garbage  
Collector

01

Parallel Garbage  
Collector

02

CMS Garbage  
Collector

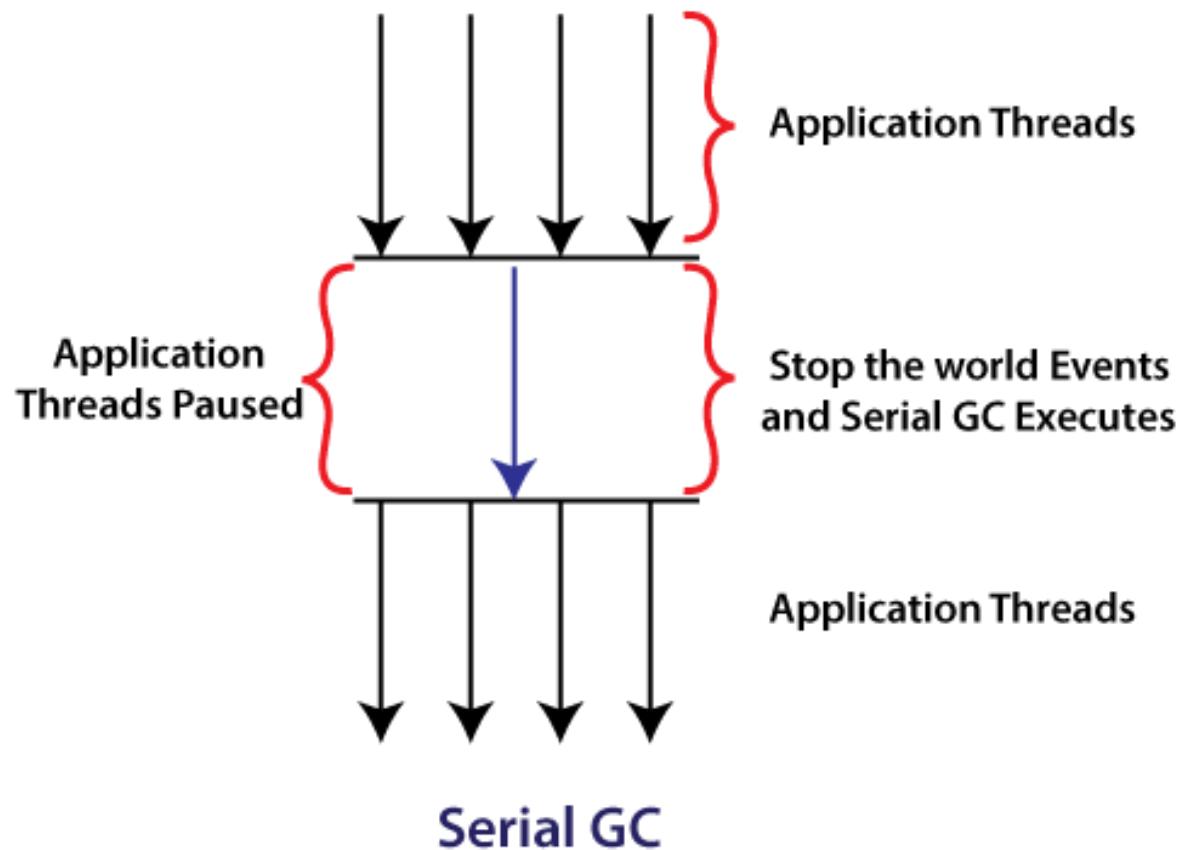
03

G1 Garbage  
Collector

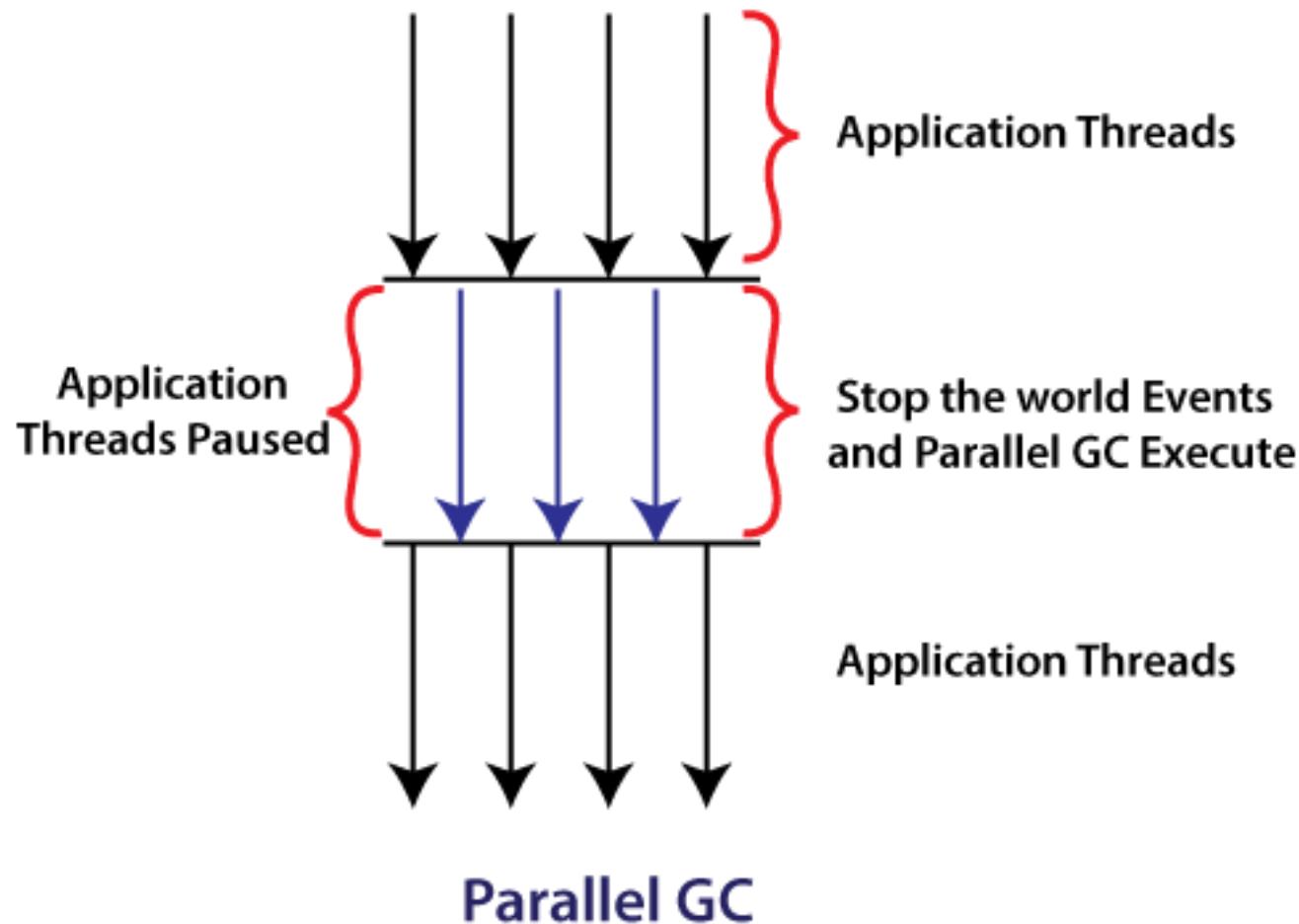
04

# Serial GC

---

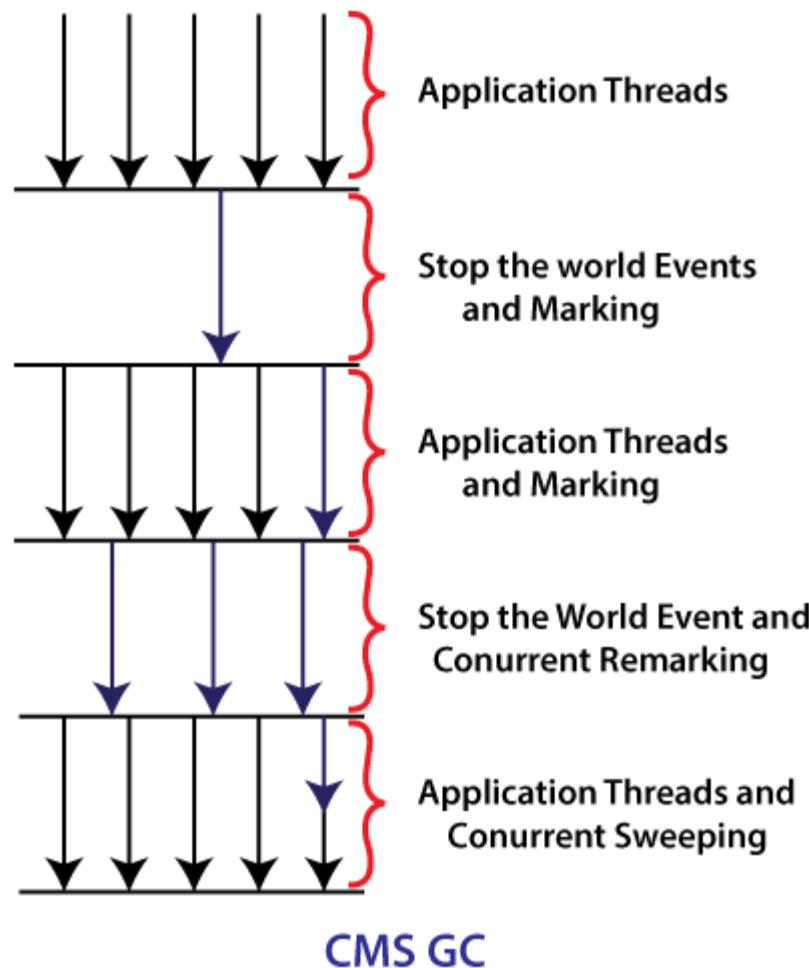


# Parallel GC



# CMS GC

---



## Garbage First (G1) Garbage Collector

---

- The G1 garbage collector is used if we have a large (more than 4GB) memory (heap space).
- It divides the heap into equal-sized (usually 1MB to 32MB) chunks, prioritizes them, and then performs the parallel garbage collection on those chunks based on the priority.

## Garbage First (G1) Garbage Collector

---

- The Eden, survivors, and old areas use this equal-sized region for the memory allocation of the objects. Apart from these memory regions, there are two more types of regions presented in the G1 GC:
- Humongous: It is used if the object sized is large.
- Available: It represents the unoccupied space.

# Garbage First (G1) Garbage Collector

E	S		H
T		S	
T	H	E	T
E	E	S	

G1GC



## Java10 GC Enhancements

---

- G1 was made the default garbage collector as of Java 9 in place of Parallel GC, as the general consensus was that low pause times are more important than higher throughput.
- JEP 304: Garbage Collector Interface
- Introducing an interface for garbage collectors will improve the modularity of the HotSpot JVM, making it much easier and simpler to add a new GC or exclude an existing one from a JDK build.

## Java10 GC Enhancements

---

- Java 9 introduced G1 (Garbage First) garbage collector.
- G1 avoids full garbage collection but in case of concurrent threads look for collection and memory is not revived fast enough, user experience is impacted.
- With Java 10, now G1 will use a fallback Full Garbage Collection.
- With this change, G1 improves its worst-case latency by using a Full GC in parallel.
- At present, G1 uses a single threaded mark-sweep-compact algorithm.
- With JEP 307, a parallel thread will start mark-sweep-compact algorithm. Number of threads can be controlled using following option.

## Java10 GC Enhancements

---

- Before Java10, the garbage collector code was fragmented in places scattered all over the HotSpot sources.
- This resulted in a few problems:
  - Anyone wanting to implement a new GC would require knowledge about all these various places, as well as how to extend the various classes for their specific needs.
  - The same knowledge was required to remove / exclude a GC at build time.
  - For anyone who wasn't familiar with the GC code, it was confusing where to find a particular piece of code for a given GC.

## Java10 GC Enhancements

---

- In Java 10, G1GC is getting a performance boost with the introduction of full parallel processing during a Full GC.
- This change won't help the best-case performance times of the garbage collector, but it does significantly reduce the worst-case latencies.
- This makes pauses for garbage collection far less stressful on application performance.
- When concurrent garbage collection falls behind, it triggers a Full GC collection.
- The performance improvement modifies the full collection so it is no longer single-threaded, which significantly reduces the time needed to do a full garbage collection.

# G1GC

---

- The Garbage First Garbage Collector (G1 GC) is a low-pause, server-style generational garbage collector for Java HotSpot VM.
- The G1 GC uses concurrent and parallel phases to achieve its target pause time and to maintain good throughput.
- When G1 GC determines that a garbage collection is necessary, it collects the regions with the least live data first (garbage first).
- G1's mechanisms to incrementally reclaim space in the heap and the pause-time control incur some overhead in both the application threads and in the space-reclamation efficiency.

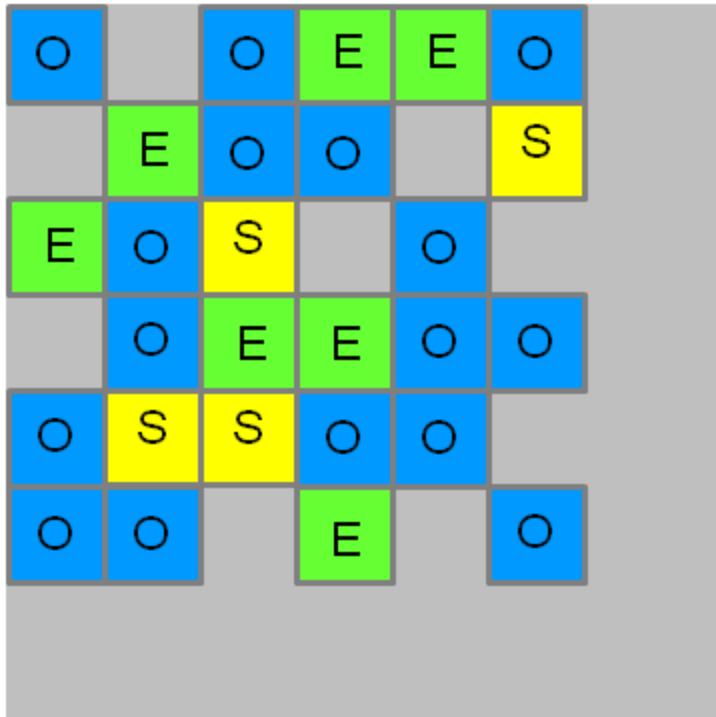
## G1GC

---

- G1 aims to provide the best balance between latency and throughput using current target applications and environments whose features include:
- Heap sizes up to ten of GBs or larger, with more than 50% of the Java heap occupied with live data.
- Rates of object allocation and promotion that can vary significantly over time.
- A significant amount of fragmentation in the heap.
- Predictable pause-time target goals that aren't longer than a few hundred milliseconds, avoiding long garbage collection pauses.

# G1GC

---



**E** Eden Space

**S** Survivor Space

**O** Old Generation

## Recommended Settings for G1GC

---

- Based on initial testing, there appears to be no significant improvement in the NameNode startup process when using G1GC rather than CMS. The following NameNode settings are recommended for G1GC in a large cluster:
- Approximately 10% more Java heap space (-XX:Xms and -XX:Xmx) should be allocated to the NameNode, as compared to CMS setup. Recommendations for setting CMS heap size are described in Configuring NameNode Heap Size.
- For large clusters (>50M files), MaxGCPauseMillis should be set to 4000.
- You should set ParallelGCThreads to 20 (default for a 32-core machine), as opposed to 8 for CMS.
- Other G1GC parameters should be left set to their default values.

## Recommended Settings for G1GC

---

- We have observed that the G1GC does not comply with the maximum heap size (-XX:Xmx) setting. For Xmx = 110 GB, we observed the following VM statistics:
- For CMS: Maximum heap (VmPeak) = 113 GB.
- For G1GC: Maximum heap (VmPeak) = 147 GB.

# Configuration Settings for G1GC

---

- To switch from CMS to G1GC, you must update the HADOOP\_NAMENODE\_OPTS settings in the hadoop-env.sh file. On the Ambari dashboard, select HDFS > Configs > Advanced > Advanced hadoop-env, then make the following changes to the HADOOP\_NAMENODE\_OPTS settings:
  - Replace -XX:+UseConcMarkSweepGC with -XX:+UseG1GC
  - Remove -XX:+UseCMSInitiatingOccupancyOnly and -XX:CMSInitiatingOccupancyFraction=####
  - Remove -XX:NewSize=#### and -XX:MaxNewSize=####
  - Add -XX:MaxGCPauseMillis=#### (optional)
  - Add -XX:InitiatingHeapOccupancyPercent=#### (optional)
  - Add -XX:ParallelGCThreads=####, if not present (optional) . The default value of this parameter is set to the number of logical processors (up to a value of 8). For more than 8 logical processors, the default value is set to 5/8th the number of logical processors.
  - The optional settings can be used to change the default value of the setting.

# Consolidate the JDK Forest into a Single Repository

---

- Java 10 simplified the JDK Forest repository into a single one in order to simplify and streamline development.
- In previous versions of Java, JDK has been broken into several repositories. Such as JDK 9 consists of eight repositories: root, nashorn, jaxp, jaxws, JDK, CORBA, hotspot, and langtools.
- In Java 10, all the eight repositories have collected into a single repository.

# Application Class Data Sharing

---

- When JVM starts it loads the classes in memory as a preliminary step.
- In case there are multiple jars having multiple classes, an evident lag appears for the first request.
- In serverless architecture, such a lag can delay the boot time which is a critical operation in such an architecture.
- Application class-data sharing concept helps in reducing the start up time of an application.
- Java has an existing CDS (Class-Data Sharing) feature.
- With Application class-data sharing, Java 10 allows to put application classes in a shared archive.
- This reduces the application startup and footprint by sharing a common class meta data across multiple java processes.

# Class Data Sharing

```
C:\Administrator: Command Prompt
C:\WINDOWS\system32>java -Xshare:dump
narrow_klass_base = 0x0000000800000000, narrow_klass_shift = 3
Allocated temporary class space: 1073741824 bytes at 0x00000008c0000000
Allocated shared space: 3221225472 bytes at 0x0000000800000000
Loading classes to share ...
Loading classes to share: done.
Rewriting and linking classes ...
Rewriting and linking classes: done
Number of classes 1216
    instance classes      = 1156
    obj array classes     =   52
    type array classes    =    8
Updating ConstMethods ... done.
Removing unshareable information ... done.
Scanning all metaspace objects ...
Allocating RW objects ...
Allocating RO objects ...
Relocating embedded pointers ...
Relocating external roots ...
Dumping symbol table ...
Relocating SystemDictionary::_well_known_klasses[] ...
Removing java_mirror ... done.
mc space:    8152 [  0.0% of total] out of     65536 bytes [ 12.4% used] at 0x0000000800000000
rw space:  3889272 [ 22.1% of total] out of  3932160 bytes [ 98.9% used] at 0x0000000800010000
ro space:  7126968 [ 40.4% of total] out of  7143424 bytes [ 99.8% used] at 0x00000008003d0000
md space:    2560 [  0.0% of total] out of     65536 bytes [  3.9% used] at 0x0000000800aa0000
od space:  6360544 [ 36.1% of total] out of  6422528 bytes [ 99.0% used] at 0x0000000800ab0000
total    : 17387496 [100.0% of total] out of 17629184 bytes [ 98.6% used]
```

# Class Data Sharing

java9ws - java9moduledemo/pom.xml - Eclipse IDE

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer    java9moduledemo/pom.xml

```
<version>3.8.1</version>
<configuration>
    <release>11</release>
</configuration>
</plugin>
<plugin>
    <!-- Build an executable JAR -->
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <archive>
            <manifest>
                <addClasspath>true</addClasspath>
                <classpathPrefix>lib/</classpathPrefix>
                <mainClass>com.boa.java9moduledemo.utilities.LoggerTest</mainClass>
            </manifest>
        </archive>
    </configuration>
</plugin>
</plugins>
</build>
</project>
```

Overview Dependencies Dependency Hierarchy Effective POM pom.xml

Problems Javadoc Declaration Console Git Staging Git Repositories

<terminated> java9moduledemo [Maven Build] C:\Program Files\Java\jdk-11.0.12\bin\javaw.exe (04-Aug-2021, 8:28:36 am – 8:28:43 am)

[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!

[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ java9moduledemo ---

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java9moduledemo ---

## Steps – Application Class Data Sharing

---

- Application Class data sharing is a 3 step process.
- Create a list of Classes to archive – Create a list welcome.lst of a class Greeting.java lying in welcome.jar using Java Launcher.
- `$java -Xshare:off -XX:+UseAppCDS -XX:DumpLoadedClassList=welcome.lst -cp welcome.jar Greeting`

## Steps – Application Class Data Sharing

---

- Create AppCDS archive – Archive a list of classes to be used for Application class data sharing.
- `$java -Xshare:dump -XX:+UseAppCDS -XX:SharedClassListFile=welcome.lst -XX:SharedArchiveFile=welcome.jsa -cp welcome.jar`

## Steps – Application Class Data Sharing

---

- Use AppCDS archive – Use AppCDS archive while using java launcher.
- \$java -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=welcome.jsa -cp welcome.jar Greeting

# Application Class Data Sharing

```
Administrator: Command Prompt
F:\java9ws\java9moduledemo>java -jar target/java9moduledemo-0.0.1-SNAPSHOT.jar
Aug 04, 2021 8:38:14 AM com.boa.java9moduledemo.utilities.LoggerTest main
INFO: Running test application.

F:\java9ws\java9moduledemo>
```

# Application Class Data Sharing

```
Administrator: Command Prompt
F:\java9ws\java9moduledemo>java -jar target/java9moduledemo-0.0.1-SNAPSHOT.jar
Aug 04, 2021 8:38:14 AM com.boa.java9moduledemo.utilities.LoggerTest main
INFO: Running test application..

F:\java9ws\java9moduledemo>java -Xshare:on target/java9moduledemo-0.0.1-SNAPSHOT.jar
Aug 04, 2021 8:38:57 AM com.boa.java9moduledemo.utilities.LoggerTest main
INFO: Running test application..

F:\java9ws\java9moduledemo>
```



# Application Class Data Sharing

```
Administrator: Command Prompt
F:\java9ws\java9moduledemo>java -jar -xlog:class+load -Xshare:on target/java9moduledemo-0.0.1-SNAPSHOT.jar
Unrecognized option: -xlog:class+load
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.

F:\java9ws\java9moduledemo>java -jar -Xlog:class+load -Xshare:on target/java9moduledemo-0.0.1-SNAPSHOT.jar
[0.011s][info][class,load] opened: C:\Program Files\Java\jdk-11.0.12\lib\modules
[0.022s][info][class,load] java.lang.Object source: shared objects file
[0.023s][info][class,load] java.io.Serializable source: shared objects file
[0.024s][info][class,load] java.lang.Comparable source: shared objects file
[0.024s][info][class,load] java.lang.CharSequence source: shared objects file
[0.024s][info][class,load] java.lang.String source: shared objects file
[0.024s][info][class,load] java.lang.reflect.AnnotatedElement source: shared objects file
[0.024s][info][class,load] java.lang.reflect.GenericDeclaration source: shared objects file
[0.025s][info][class,load] java.lang.reflect.Type source: shared objects file
[0.025s][info][class,load] java.lang.Class source: shared objects file
[0.025s][info][class,load] java.lang.Cloneable source: shared objects file
[0.025s][info][class,load] java.lang.ClassLoader source: shared objects file
[0.025s][info][class,load] java.lang.System source: shared objects file
[0.025s][info][class,load] java.lang.Throwable source: shared objects file
[0.026s][info][class,load] java.lang.Error source: shared objects file
[0.026s][info][class,load] java.lang.ThreadDeath source: shared objects file
[0.026s][info][class,load] java.lang.Exception source: shared objects file
[0.026s][info][class,load] java.lang.RuntimeException source: shared objects file
[0.026s][info][class,load] java.lang.SecurityManager source: shared objects file
[0.027s][info][class,load] java.security.ProtectionDomain source: shared objects file
[0.027s][info][class,load] java.security.AccessControlContext source: shared objects file
[0.028s][info][class,load] java.security.SecureClassLoader source: shared objects file
[0.028s][info][class,load] java.lang.ReflectiveOperationException source: shared objects file
[0.029s][info][class,load] java.lang.ClassNotFoundException source: shared objects file
[0.030s][info][class,load] java.lang.LinkageError source: shared objects file
[0.030s][info][class,load] java.lang.NoClassDefFoundError source: shared objects file
[0.030s][info][class,load] java.lang.ClassCastException source: shared objects file
```

## Tool Enhancements

---

- Java 21 offers better tools for creating new processes, such as ‘Runtime.exec’ and ‘ProcessBuilder’.
- This feature tracks or logs these operations, especially by monitoring the logger ‘java.lang.ProcessBuilder’.
- For example, to test a ping command, use ‘ProcessBuilder processBuilder’.
- The process details are continuously logged based on the logger’s level, which can be either DEBUG or TRACE.
- For example, if the log level is set to TRACE, the procedure will begin.

## Java Emoji Compatibility Tools

---

- Java has added a method to the ‘java.lang.Character’ class that works with different emoji attributes as described by the Unicode Standard UTS#51.
- This function may detect if a given code leads to a unique integer that represents a character as an emoji.
- Whether a code point can have an emoji modifier applied, such as the hand emoji with varying skin tones.



## What the heck are Sealed Classes in Java?

---

- Think of **sealed classes** (introduced in Java 17) as **VIP-only clubs**.
- You're the “parent class”, and you decide **exactly who is allowed inside** — meaning which classes are allowed to extend you.



## Why Do We Use It?

---

- Sealed classes shine when you want:
- **Controlled inheritance**
- **Exhaustive switch handling** (pattern matching!)
- **Better domain modeling** (e.g., Payment → CreditCard / UPI / Wallet)
- **Security** — no random “unknown subclasses”



## How To Declare a Sealed Class

---

- public sealed class Vehicle permits Car, Bike, Truck {
- }
- sealed → tells Java the inheritance is restricted.
- permits → lists who can extend it.



## Key Rules to Remember

---

- ✓ A sealed class **must specify** permitted classes
  - ✓ Child classes **must declare** final, sealed, or non-sealed
  - ✓ All permitted subclasses must be **in the same module** (or same package, if not using modules)
  - ✓ Good for domain-driven design, algebraic hierarchies, finite types



# Super Clean Summary

---

Keyword	Meaning
sealed	Restricts which classes can extend it
permits	Lists allowed subclasses
final	No further extension
non-sealed	Open for extension
sealed (child)	Further restricted hierarchy



# Java 21 — Deprecation & Removals (The Chill but Clear Version)

---

- ♦ 1. **Thread.stop() family — AGAIN**
- Even though Thread.stop, Thread.suspend, resume have been deprecated since forever, Java 21 continues tightening restrictions.
- Not new in 21, but still part of the “final cleanup list”.



# Java 21 — Deprecation & Removals (The Chill but Clear Version)

---

- ♦ 2. Finalization (→ Marked for Removal)
- Java is killing finalizers.
- In Java 21:
- Finalization is **disabled by default** (major change!)
- You can re-enable with:
- --finalization=enabled



# Java 21 — Deprecation & Removals (The Chill but Clear Version)

---

- **◆ 3. Security Manager (Full Removal Path)**
- The Security Manager is deprecated *for removal*.
- Affected:
- `java.lang.SecurityManager`
- `java.lang.System::setSecurityManager`
- `java.lang.ClassLoader::setDefaultAssertionStatus`
- No one really used it in containerized/cloud world anyway.



# Java 21 — Deprecation & Removals (The Chill but Clear Version)

---

- **4. RMI Activation System (Marked for Removal)**
- The entire RMI activation group is deprecated for removal.
- Classes affected:
- `java.rmi.activation.*`
- Activation-related registry APIs
- Reason: no one uses RMI activation since the early 2000s.



# Java 21 — Deprecation & Removals (The Chill but Clear Version)

---

- ◆ **5. Nashorn JavaScript Engine (Still gone)**
- Removed in Java 15, but Java 21 adds deprecation warnings in tools relying on it.
- ◆ **6. Applets — completely dead (already removed but referenced)**
- All applet APIs remain gone.



# Java 21 — Actual Removals (The Chill but Clear Version)

---

- **X 1. The Java EE & CORBA modules (already removed since Java 11, but Java 21 fully strips leftover references)**
- Modules removed earlier but confirmed gone:
- java.xml.ws
- java.xml.bind
- java.activation
- java.transaction
- java.corba
- java.se.ee
- No changes in 21 here, but worth knowing if coming from older JDKs.



## Java 21 — Actual Removals (The Chill but Clear Version)

---

- **✗ 2. jdk.jfr.internal unexported APIs**
- These internal packages are removed or hidden.



# Java 21 — Actual Removals (The Chill but Clear Version)

---

-  **Feature Deprecation (Language-Level)**
-  **2. Primitive wrapper constructors**
- Like:
- `new Integer(5)`
- `new Boolean(true)`
- These have been deprecated since Java 9, and 21 keeps them deprecated-for-removal.
- Use:
- `Integer.valueOf(5);`



# Big Deprecation Highlights in Java 21 (Quick Table)

---

Area	What Happened
Finalization	Disabled by default, deprecated for removal
Security Manager	Deprecated for removal
RMI Activation	Deprecated for removal
FileInputStream.finalize	deprecated
Object.finalize	deprecated
Swing/JavaFX applet stuff	legacy, gone
Nashorn engine	long gone
Old Java EE modules	not coming back



## Bonus: JVM Flags Deprecated in Java 21

---

- **Deprecated:**
- -Xverify options
- -Xcomp (compilation-only mode)
- UseCSR, UseCMG, old GC tuning flags
- JNI global ref constraints
- **Removed:**
- HotSpot internal debugging flags that existed only for experimentation



# Java 21 Performance Improvements — The Good Stuff

---

- Java 21 = More speed, less memory, smarter runtime, faster startup, smoother GC.

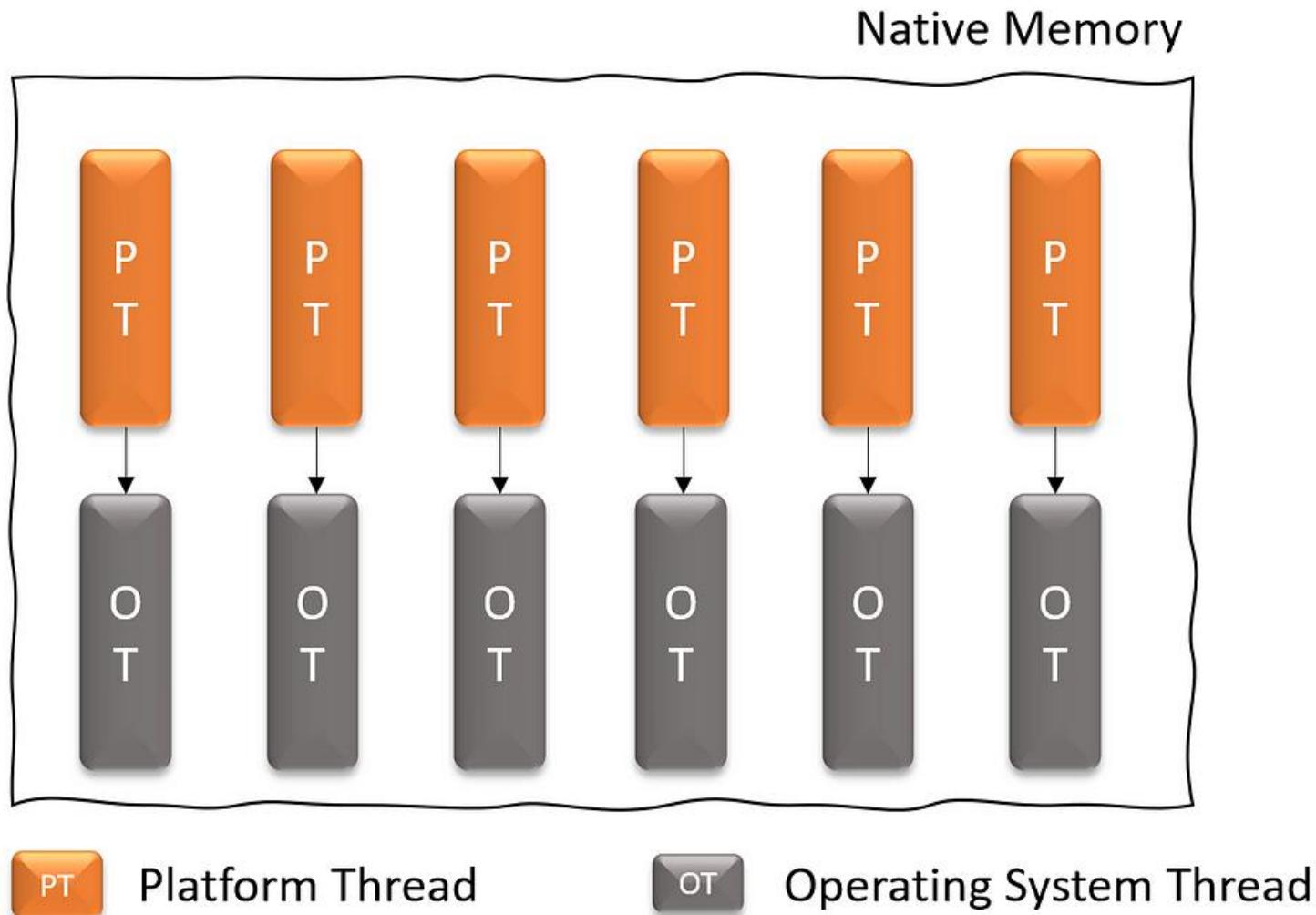


# Java 21 Performance Improvements — The Good Stuff

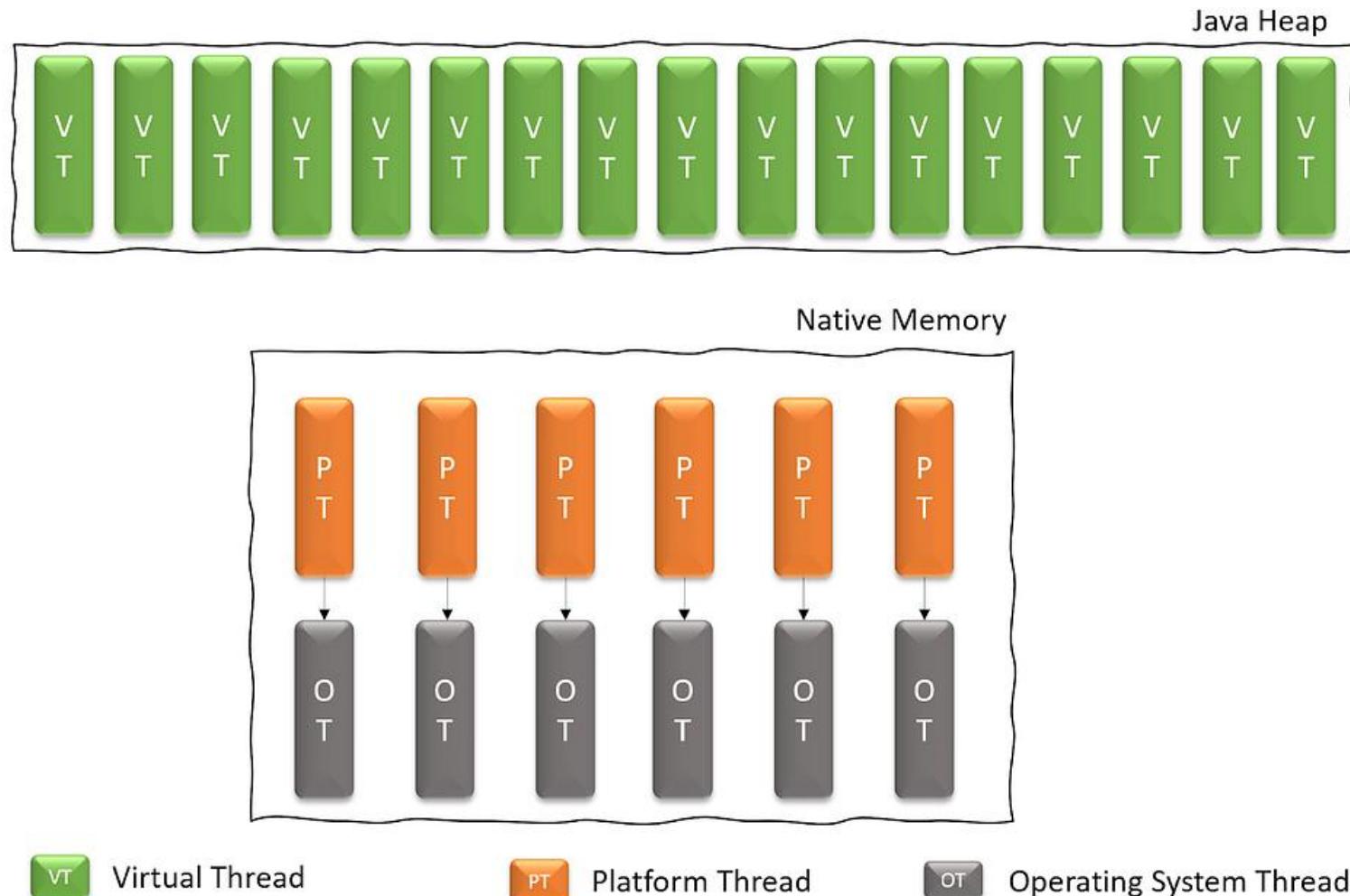
---

-  **1. Virtual Threads (Project Loom) → Massive Concurrency Boost**
- This is *the* superstar of Java 21.
- **Why it's huge?**
  - Virtual threads = **near-zero memory cost**
  - You can spawn **millions** of them (no joke)
  - Perfect for **web servers, microservices, Kafka consumers, database IO**, etc.
- **Real-world impact:**
  - **20x–200x more throughput** for high-concurrency IO workloads
  - **No more pain** with blocking APIs (JDBC, HTTP, S3, Redis)
  - `try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
 executor.submit(() -> httpCall());  
}`

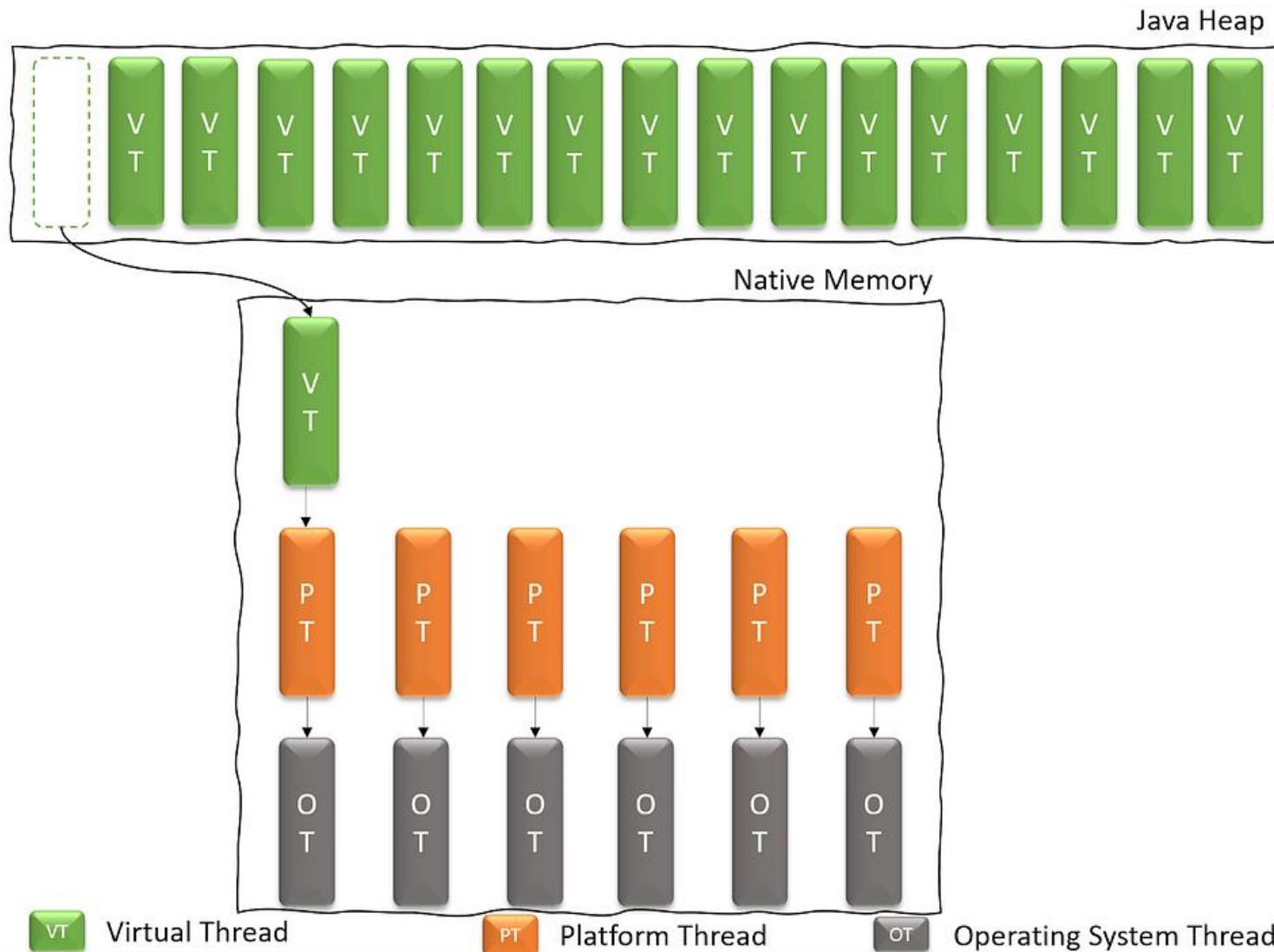
# Platform Thread



# Virtual Thread



# Virtual Threads are mapped to platform threads when it does real work



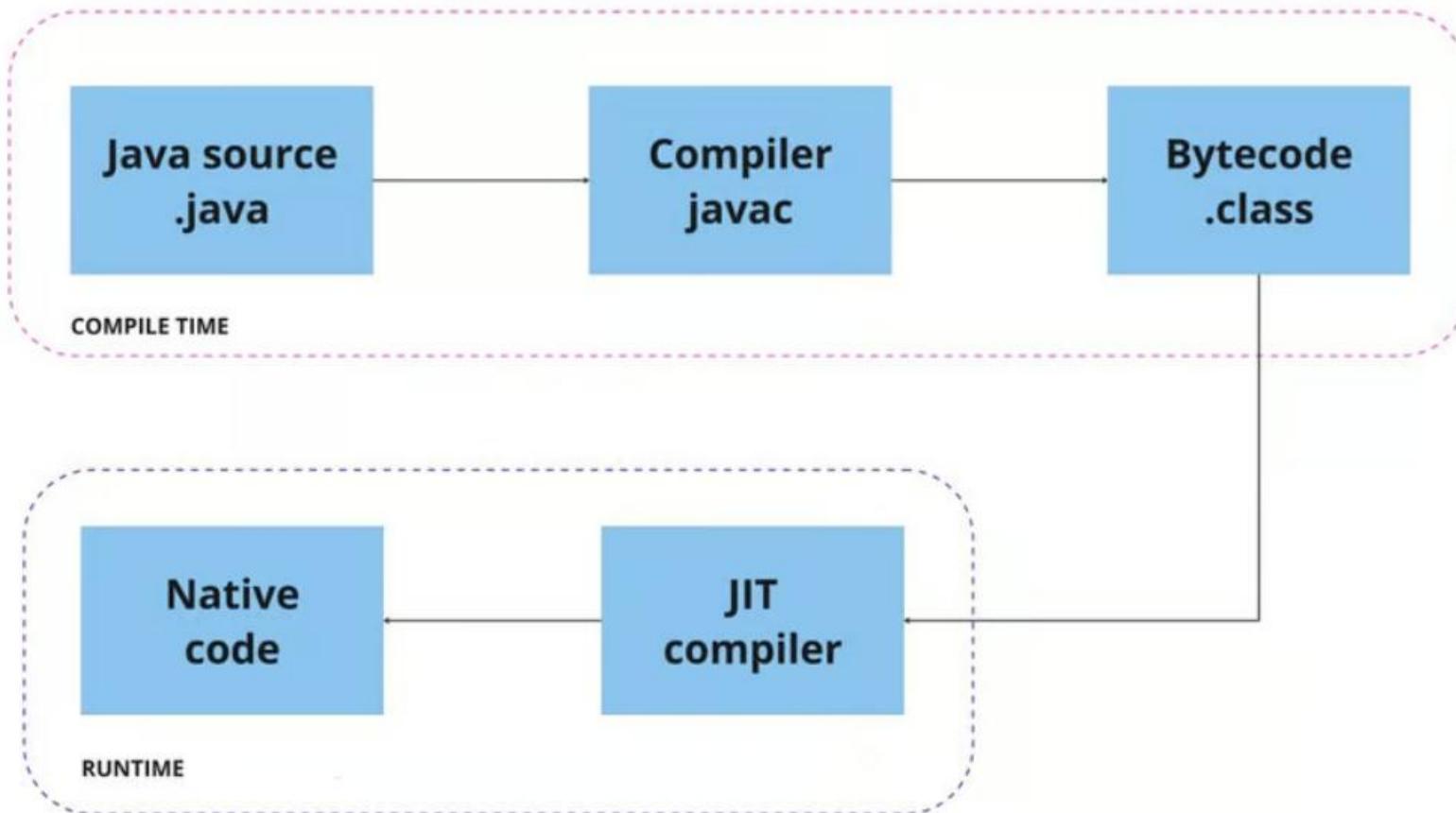


## 2. JIT / JIT Warmup Improvements (C2 & C1)

---

- **Improvements include:**
  - Better lock coarsening
  - Escape analysis enhancements
  - Faster allocation paths
  - Improved scalar replacement
  - Reduced inline cache misses
  - Faster deoptimization
- What it means for you:
  - **Less warm-up time**
  - **Better peak performance**
  - **More stable throughput** on microservices

# JIT Compiler



# The Graal JIT Compiler – It's Written in Java

---

- The OpenJDK implementation of the JVM contains two conventional JIT-compilers – the Client Compiler (C1) and the Server Compiler (C2 or Opto).
- The Client Compiler is optimized for faster operation and less optimized code output, making it ideal for desktop applications where extended JIT-compilation pauses can interrupt user experience.
- Conversely, the Server Compiler is engineered to spend more time producing highly optimized code, making it suitable for long-running server applications.

# The Graal JIT Compiler – It's Written in Java

---

- The two compilers can be used in tandem through "tiered compilation".
- Initially, the code is compiled through C1, followed by C2 if execution frequency justifies the additional compilation time.
- Developed in C++, C2, despite its high-performance characteristics, has inherent downsides.
- C++ is an unsafe language; therefore, errors in the C2 module could cause the entire VM to crash.
- The complexity and rigidity of the inherited C++ code have also resulted in its maintenance and extendibility becoming a significant challenge.

# The Graal JIT Compiler – It's Written in Java

---

- Unique to Graal, this JIT-compiler is developed in Java.
- The compiler's main requirement is accepting JVM bytecode and outputting machine code – a high-level operation that doesn't require a system-level language like C or C++.

# The Graal JIT Compiler – Benefits

---

- Improved safety: Java's garbage collection and managed memory approach eliminate the risk of memory-related crashes from the JIT compiler itself.
- Easier maintenance and extension: The Java codebase is more approachable for developers to contribute to and extend the capabilities of the JIT compiler.
- Portability: Java's platform independence translates to the Graal JIT compiler potentially working on any platform with a Java Virtual Machine.

# The Graal JIT Compiler – Benefits

---

Metric/runtime	GraalVM CE with C2 JIT	Oracle GraalVM Native Image	
Memory Usage (max RSS)	1,029 MB	641 MB	-38% lower
Peak throughput	11,066 req/s	11,902 req/s	+8% higher
Throughput per memory	12,488 req/(GB*s)	18,569 req/(GB*s)	+49% better
Tail latency (P99)	7.2ms	5.15ms	-28% lower
Startup	7,090ms	210ms	34x faster

## The experimental Java-based JIT compiler

---

- Graal, a Java-based JIT compiler, is the basis of the experimental Ahead-of-Time (AOT) compiler introduced in JDK 9.
- Enabling it to be used as an experimental JIT compiler is one of the initiatives of Project Metropolis.
- It is the next step in investigating the feasibility of a Java-based JIT for the JDK.

## Ahead-of-Time Compilation

---

- While doing profiling manually or manually check thread or memory dump from JVM we might need to perform compilation ahead of time.
- This feature is enabled after JAVA9.
- After Java9 we have the option to convert some of the classes or libraries to compiled code before the start of the application.

# Ahead-of-Time Compilation

---

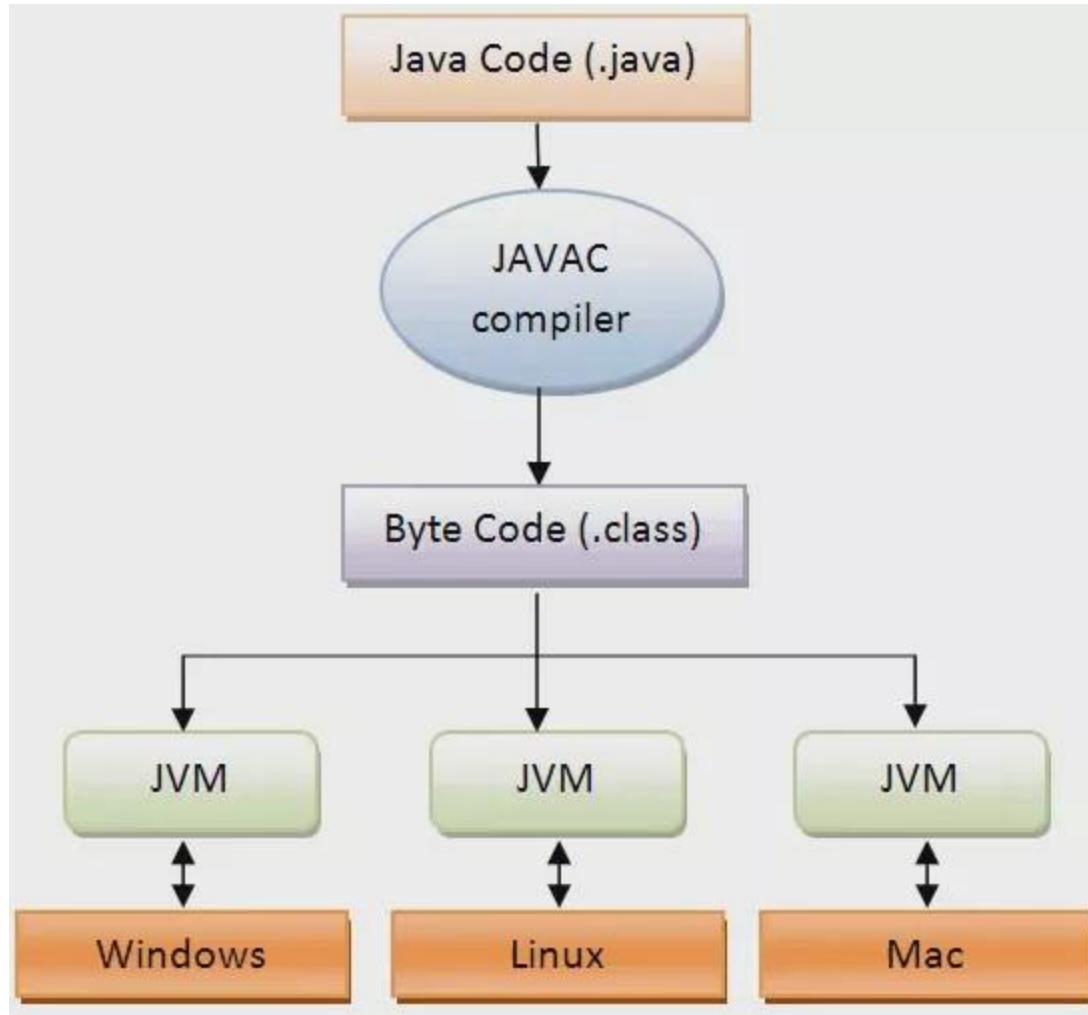
- Compile class before:
  - Before we can use the AOT compiler, we need to compile the class with the Java compiler:
  - `javac <classname>.java`
- Pass the class to the AOT compiler:
  - We then pass the resulting `<classname>.java` to the AOT compiler, which is located in the same directory as the standard Java compiler.
  - `jaotc --output <classname>.so <classname>.class`

## Ahead Of Time vs Just In Time in Java

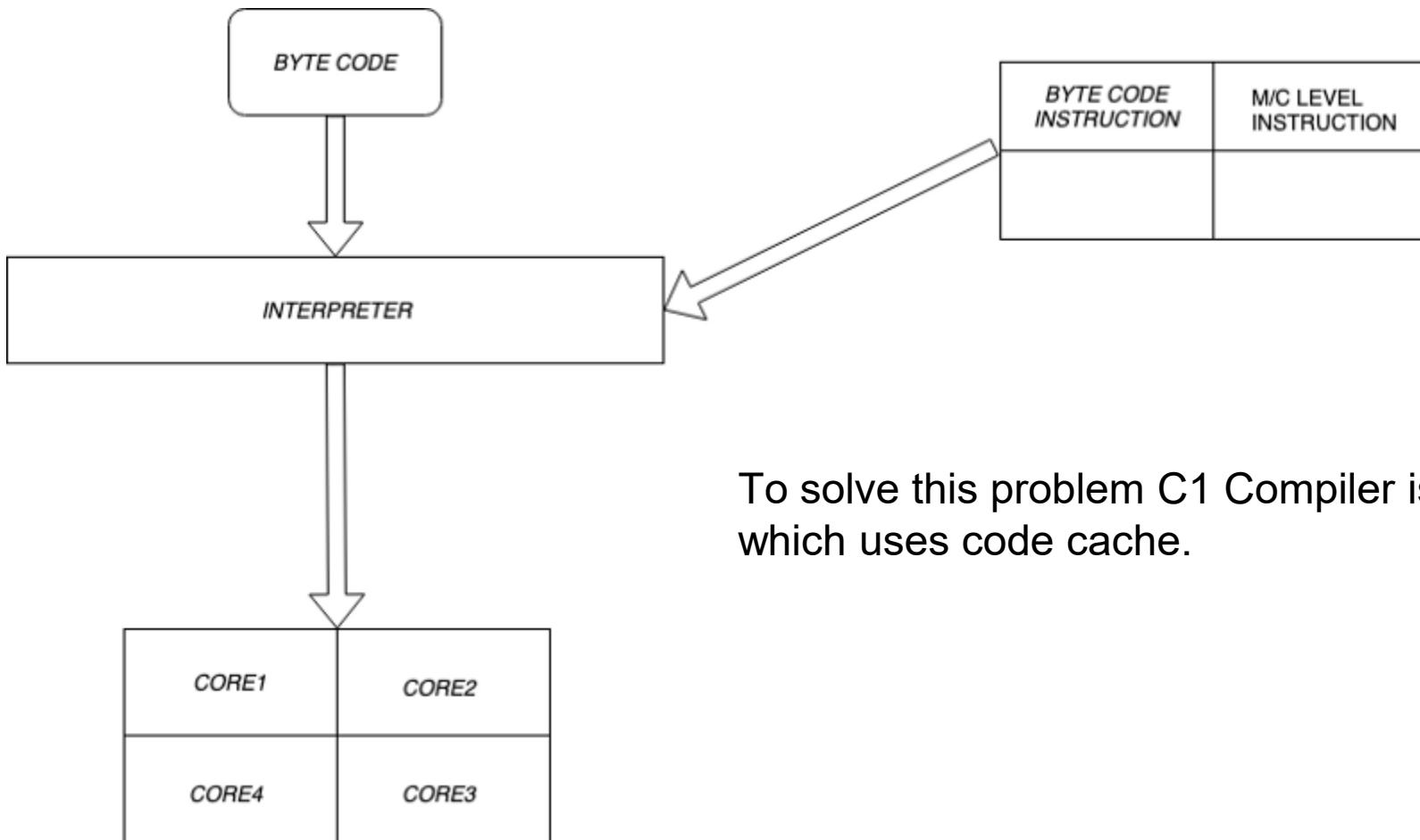
---

- How Java works:
- Java compiles the code and converts into byte code(.class)
- JVM interprets the byte codes and converts into machine level codes so that it can run on any machine

# Ahead Of Time vs Just In Time in Java

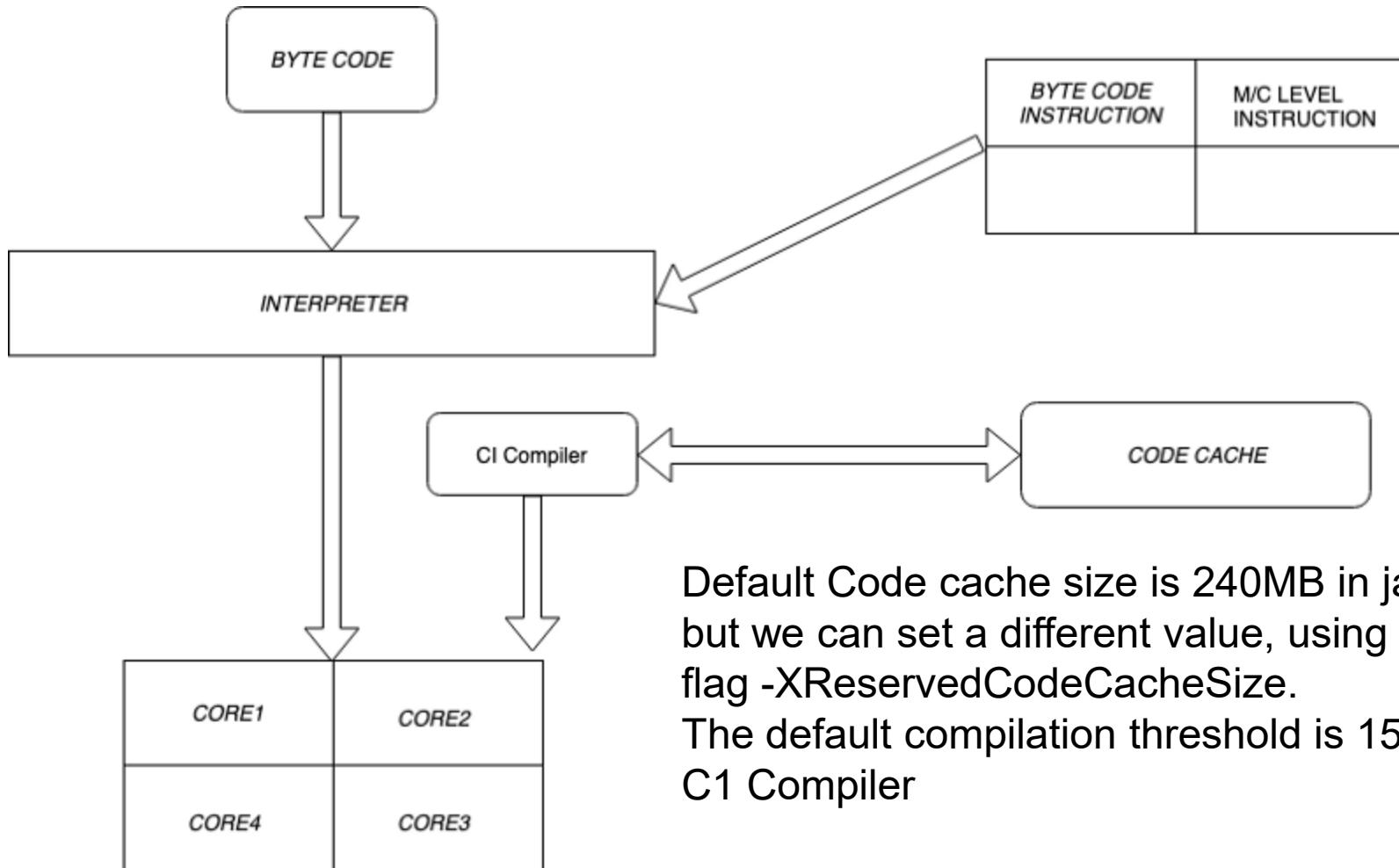


# Ahead Of Time vs Just In Time in Java



To solve this problem C1 Compiler is used, which uses code cache.

# C1 Compiler



## C1 Compiler

---

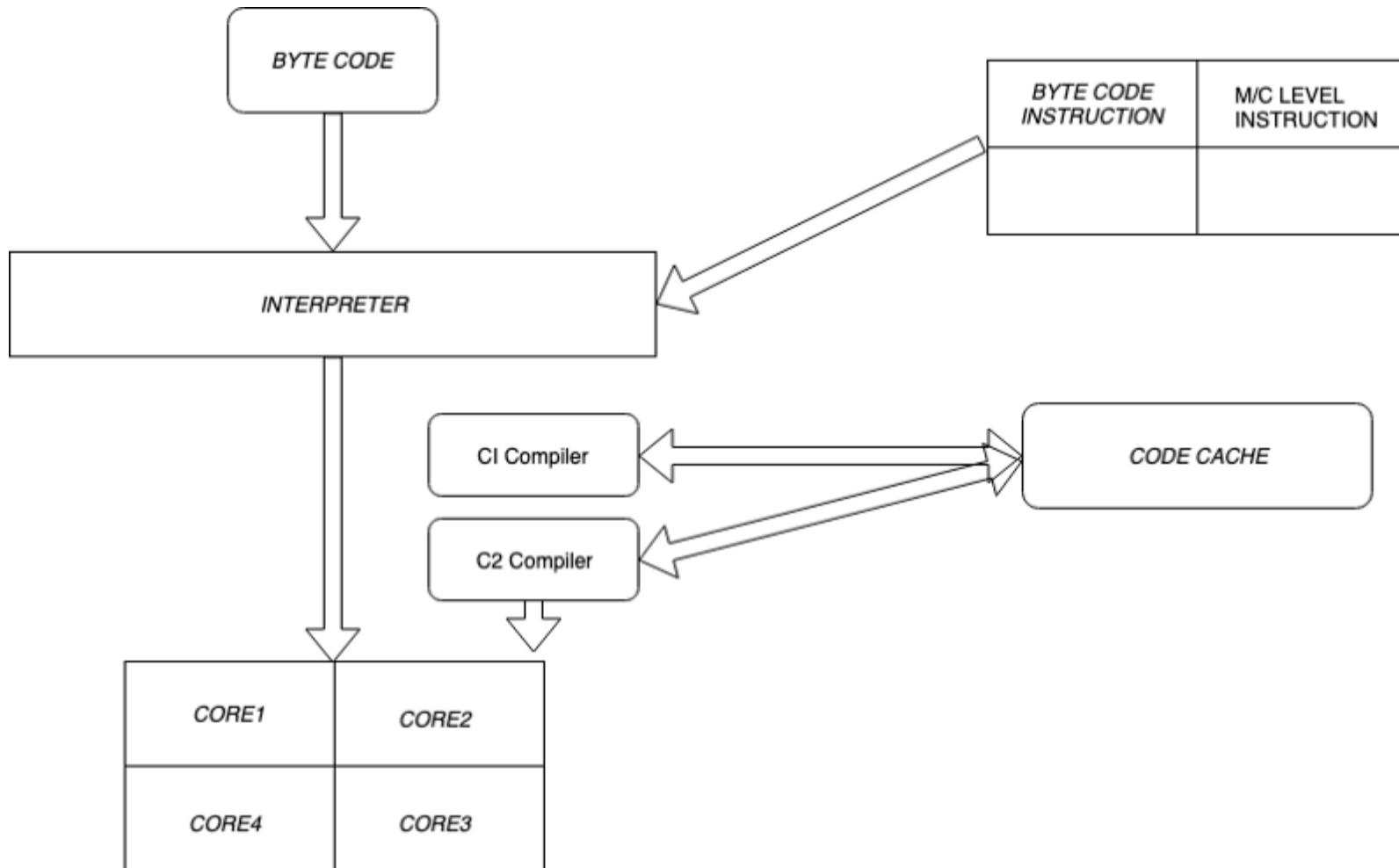
- When interpreter interprets the byte code it uses counter how many times the same byte code gets converted to machine code which is saved in the code cache.
- When the counter reaches the threshold then C1 compiler compiles the codes and save in code cache so that when same byte codes get executed it will get from code cache.
- Code cache is a memory area separate from the JVM heap that contains all the JVM bytecode for a method compiled down to native code, each called a nmethod1.
- This is where the JIT compiled methods are kept.

## C2 Compiler:

---

- After some time when JVM runs for some time, it's start collecting statistics in the background how the code is being executed called code profiling.
- It creates control flow graphs(code paths). It tries to find out hottest code paths once it has enough statistics then JVM asked for C2 compiler to perform optimizations on the hottest code paths.
- It also stores optimized code in Code Cache.

# C2 Compiler:



## C2 Compiler:

---

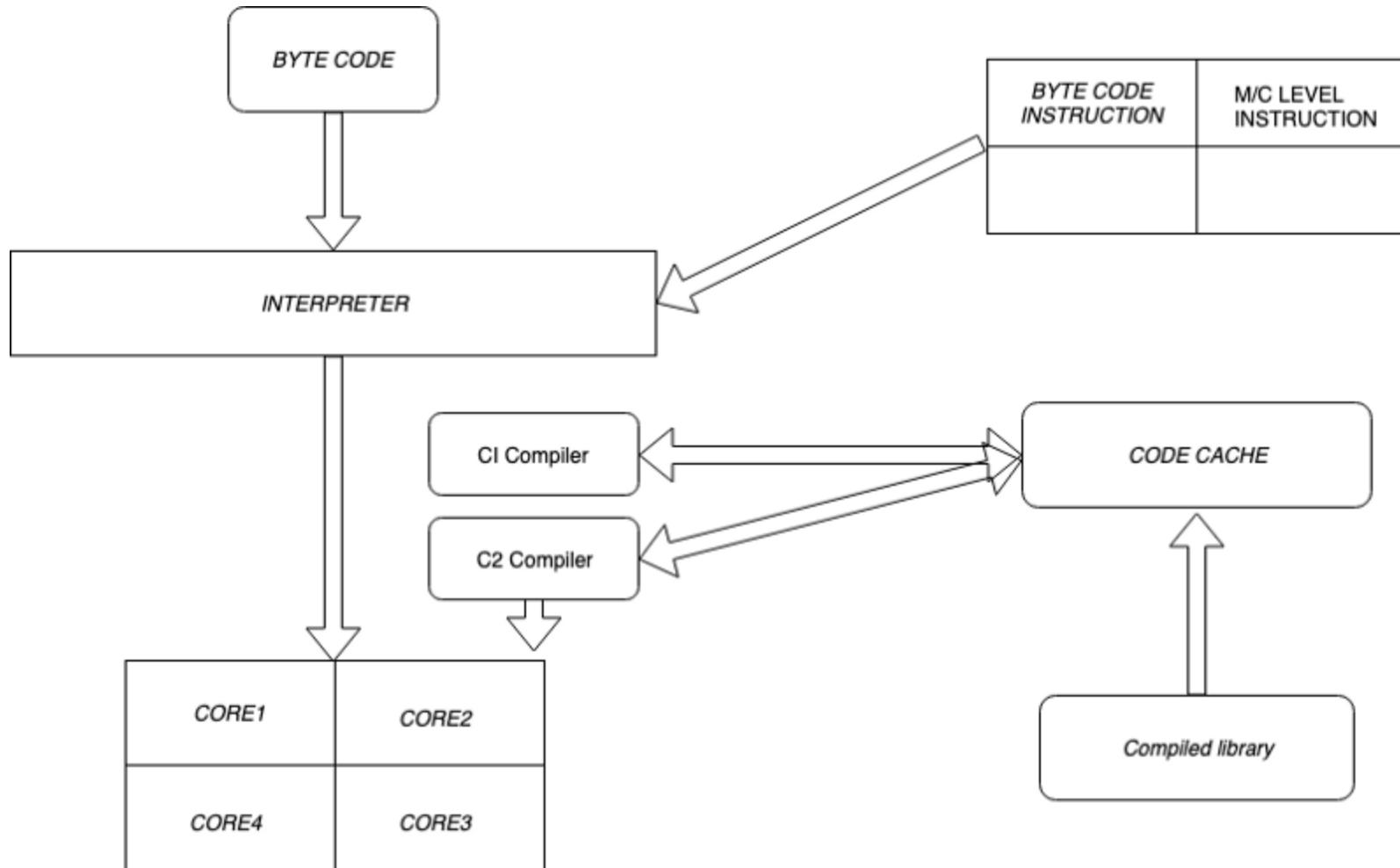
- Optimizations Perform by C2:
- Dead Code
- Escape Analysis
- Loops
- Methods Inlining
- Null check Elimination

## More about C1 and C2:

---

- Two compilers, C1, and C2 run in parallel and keep on optimizing the code
- C1 is preferred for the client application and C2 is preferred for long running server applications.
- Tiered compilation combines the best features of both compilers.
- Client-side compilation yields quick startup time and speedy optimization, while server-side compilation delivers more advanced optimizations later in the execution cycle.
- When the Code Cache is constrained (its usage approaches or reaches the ReservedCodeCacheSize), to compile more methods, the JIT must first throw out some already compiled methods.
- Discarding compiled methods are known as Code Cache flushing.
- In JAVA 7 we have the option to select to both the compiler.
- In JAVA 8 both are available by default.

# Ahead-of-Time Compilation



# Ahead-of-Time Compilation

---

- Running the Program
- We can then execute the program while running the program we need to use the flag -XX: AOTLibrary to tell the JVM for AOT compiled class.
- `java -XX:AOTLibrary=./<classname>.so <classname>`
- `java -XX:+UnlockExperimentalVMOptions -XX:AOTLibrary=com/boa/banking/Test.so com.boa.banking.Test`
- We can also see the library was loaded by adding -XX:+PrintAOT as a JVM argument.

## Ahead-of-Time Compilation

---

- Javac com\boa\banking\Test.java
- jaotc --output com\boa\banking\Test.so  
com.boa.banking.Test
- java -XX:+UnlockExperimentalVMOptions -  
XX:+PrintAOT -  
XX:AOTLibrary=com/boa/banking/Test.so  
com.boa.banking.Test



## 3. Garbage Collector Improvements (G1, ZGC, Shenandoah)

---

- **G1 GC in Java 21:**
- Smarter **region pinning**
- Faster **remembered set scanning**
- Improved **pause predictability**
- Better pre-touch performance
- String Dedup improvements
- Result:
- Shorter pauses + stable 99th percentile latency

## ◆ ZGC in Java 21:

---

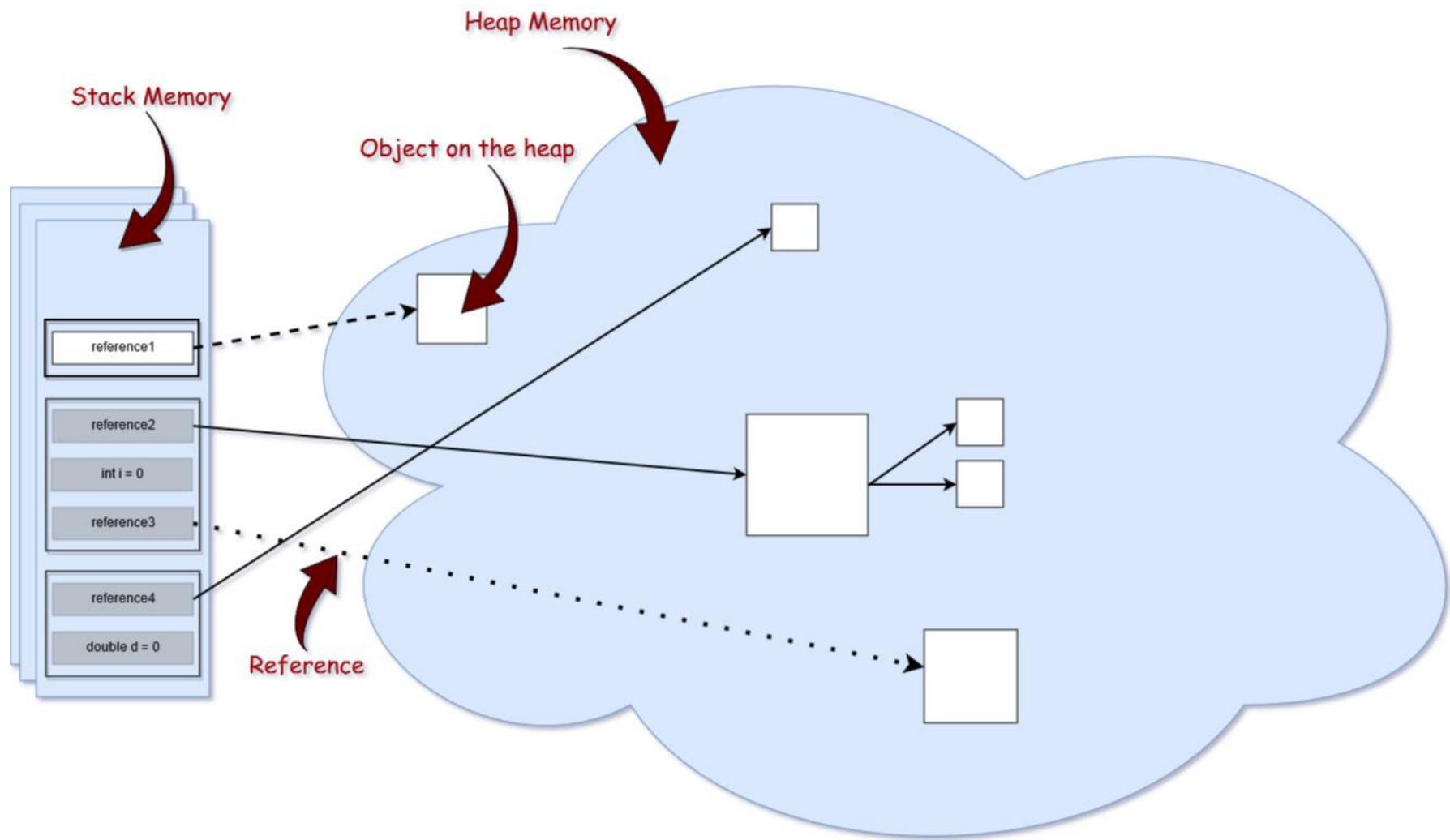
- Sub-millisecond pauses (always)
- Much faster class unloading
- Reduced memory overhead
- Compressed object pointers enabled
- Better heap compaction
- ZGC is now **production-ready for any size heap.**
- Perfect for:
  - High-frequency trading
  - Real-time microservices
  - Kafka processing
  - In-memory DBs

## ◆ Shenandoah GC:

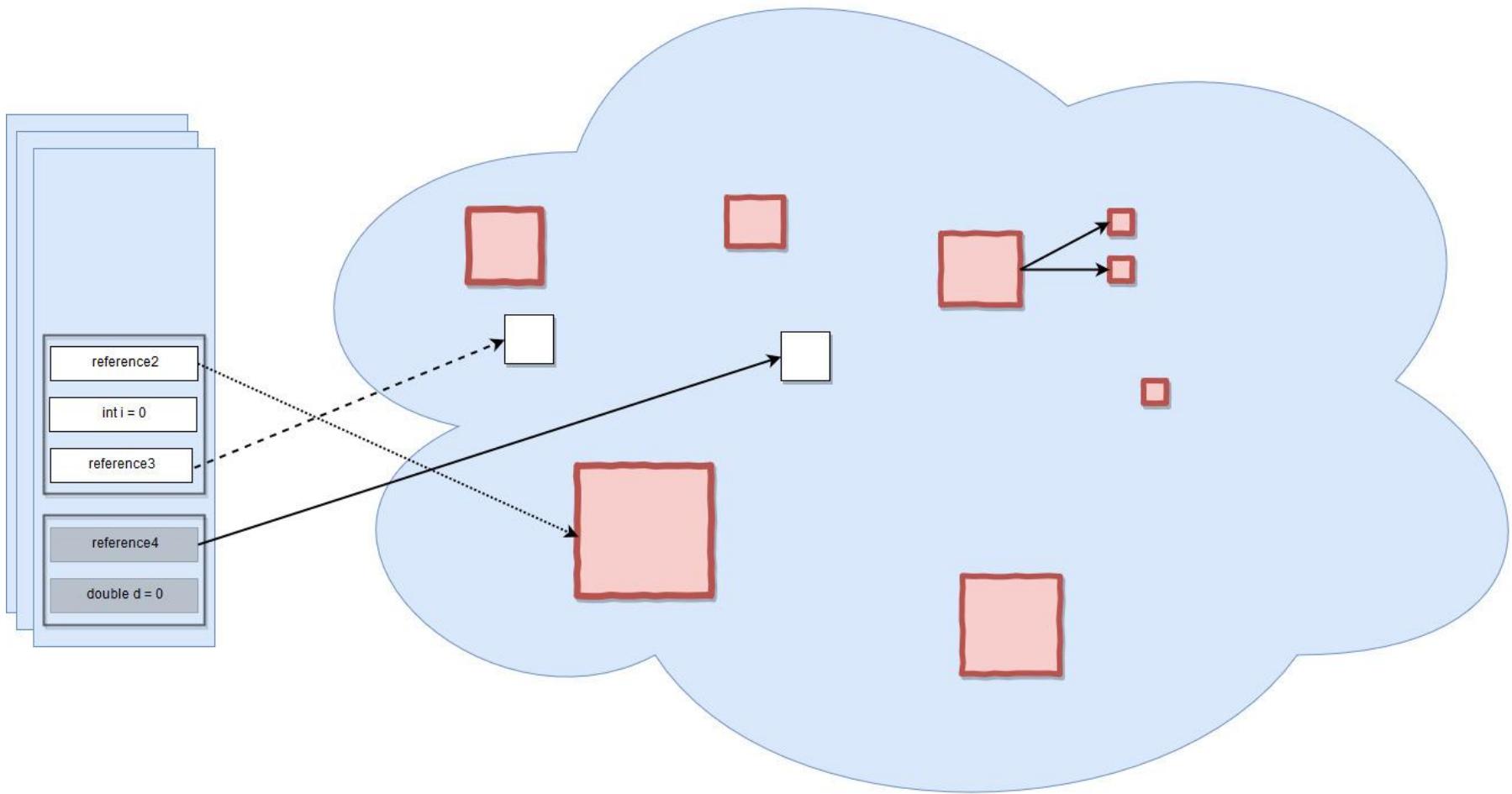
---

- Improved load barriers
- Better concurrent marking
- Leaner memory footprint

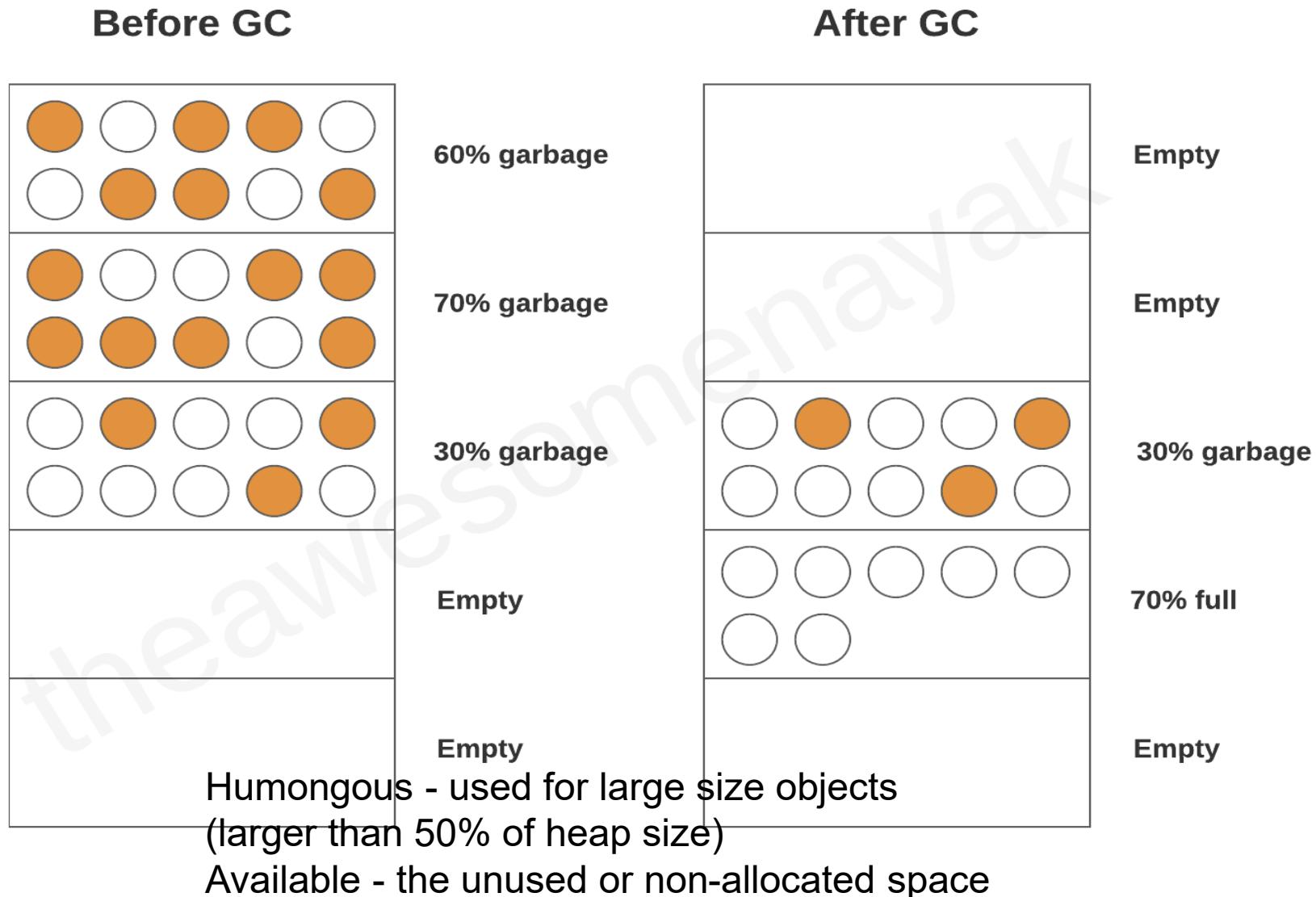
# MEMORY MANAGEMENT



# MEMORY MANAGEMENT GC



# MEMORY MANAGEMENT G1GC



# Epsilon Garbage Collector

---

- Epsilon is a do-nothing (no-op) garbage collector that was released as part of JDK 11.
- It handles memory allocation but does not implement any actual memory reclamation mechanism.
- Once the available Java heap is exhausted, the JVM shuts down.
- It can be used for ultra-latency-sensitive applications, where developers know the application memory footprint exactly, or even have (almost) completely garbage-free applications.
- **Usage of the Epsilon GC in any other scenario is otherwise discouraged.**

# Shenandoah

---

- Shenandoah is a new GC that was released as part of JDK 12.
- Shenandoah's key advantage over G1 is that it does more of its garbage collection cycle work concurrently with the application threads.
- G1 can evacuate its heap regions only when the application is paused, while Shenandoah can relocate objects concurrently with the application.
- Shenandoah can compact live objects, clean garbage, and release RAM back to the OS almost immediately after detecting free memory.
- Since all of this happens concurrently while the application is running, Shenandoah is more CPU intensive.

## Shenandoah

---

- The JVM argument to use the Epsilon Garbage Collector is `-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC`.

## ZGC

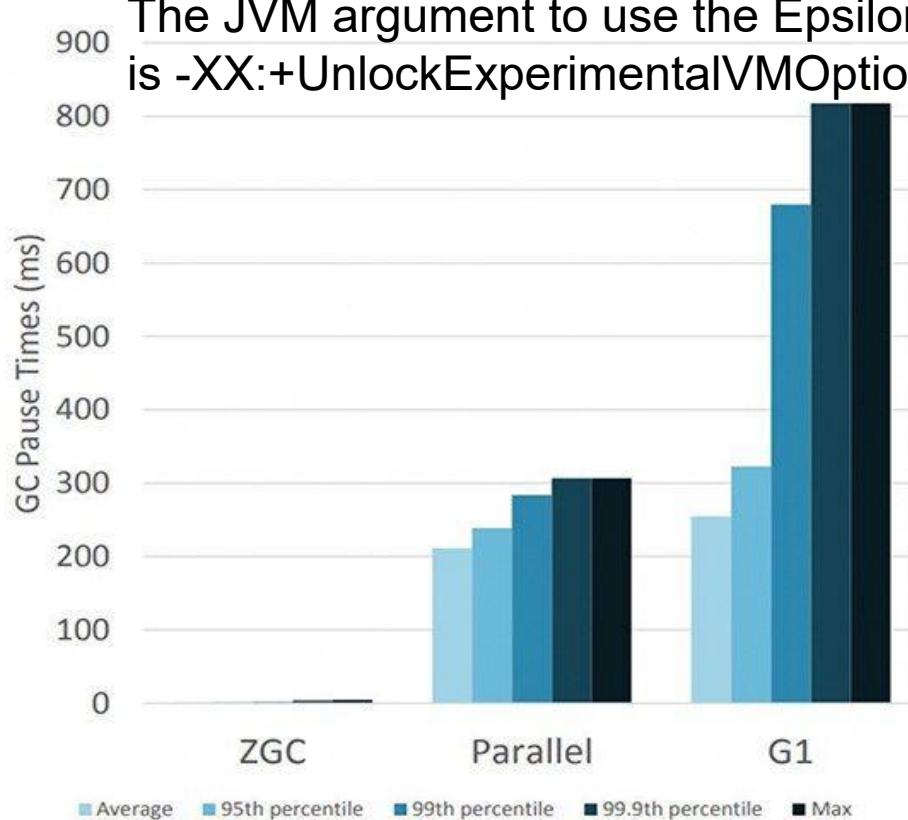
---

- ZGC is another GC that was released as part of JDK 11 and has been improved in JDK 12.
- It is intended for applications which require low latency (less than 10 ms pauses) and/or use a very large heap (multi-terabytes).
- The primary goals of ZGC are low latency, scalability, and ease of use.
- To achieve this, ZGC allows a Java application to continue running while it performs all garbage collection operations.
- By default, ZGC uncommits unused memory and returns it to the operating system.

# ZGC

- Thus, ZGC brings a significant improvement over other traditional GCs by providing extremely low pause times (typically within 2ms).

The JVM argument to use the Epsilon Garbage Collector is `-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`.



## How to Select the Right Garbage Collector

---

- Serial - If the application has a small data set (up to approximately 100 MB) and/or it will be run on a single processor with no pause-time requirements
- Parallel - If peak application performance is the priority and there are no pause-time requirements or pauses of one second or longer are acceptable
- CMS/G1 - If response time is more important than overall throughput and garbage collection pauses must be kept shorter than approximately one second
- ZGC - If response time is a high priority, and/or you are using a very large heap

# Removed GC Options(JEP 214)

---

- Deprecated in JDK 8 (JEP 173)

DefNew + CMS	:	-XX:-UseParNewGC -XX:+UseConcMarkSweepGC
ParNew + SerialOld	:	-XX:+UseParNewGC
ParNew + iCMS	:	-Xincgc
ParNew + iCMS	:	-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC
DefNew + iCMS	:	-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC -XX:-UseParNewGC
CMS foreground	:	-XX:+UseCMSCompactAtFullCollection
CMS foreground	:	-XX:+CMSFullGCsBeforeCompaction
CMS foreground	:	-XX:+UseCMSCollectionPassing

## Diagnostic Tools

---

- Java Mission Control
- Java Flight Recorder
- <https://docs.oracle.com/javase/9/troubleshoot/diagnostic-tools.htm#JSTGD148>



## 4. String, Collections & I/O Performance Boosts

---

- **Improvements:**
- Faster UTF-8 encoding/decoding
- Optimized StringBuilder & StringConcatFactory
- Better inlining for small collections
- Compact city-hash style improvements
- Faster Files.readString(), Files.writeString()
- **Result:**
- Anything involving JSON, HTTP, logs, DB serialization  
→ gets faster.



## 5. Pattern Matching + Records + Sealed Classes = Compiler Optimizations

---

- **Optimizations**
- The compiler uses your domain model to optimize branching.
- Example:
- `switch (vehicle) {`
- `case Car c -> ...`
- `case Truck t -> ...`
- `case Bike b -> ...`
- `}`
- **JVM knows all subclasses → fewer checks → faster dispatch.**

## 5. Pattern Matching + Records + Sealed Classes = Compiler Optimizations

---

-  **7. Better Vector API (Incubator)**
  - The vector API is *crazy fast* for:
  - AI/ML workloads
  - Hashing
  - Encryption
  - Image processing
- Uses **SIMD instructions** directly from CPU.
- Example:
  - `FloatVector a = FloatVector.fromArray(SPECIES, arr, 0);`
  - `FloatVector b = FloatVector.broadcast(SPECIES, 3.14f);`
  - `FloatVector result = a.mul(b);`
- Blazing fast.



## 5. Pattern Matching + Records + Sealed Classes = Compiler Optimizations

---

-  **8. Cryptography Speed-ups**
  - Java 21 has optimized:
  - AES/GCM performance
  - SHA-3 hardware acceleration
  - TLS handshake speed
- Native SSLEngine optimizations
  - Great for:
  - Banking apps
  - JWT auth
  - HTTPS microservices



## 5. Pattern Matching + Records + Sealed Classes = Compiler Optimizations

---

-  **9. Memory Efficiency Enhancements**

- Smaller thread stacks (especially virtual threads)
- Improved arena allocation
- Better escape analysis → fewer heap objects
- Reduced class metadata footprint



## 5. Pattern Matching + Records + Sealed Classes = Compiler Optimizations

---

-  **10. Foreign Function & Memory API (FFM) → Faster Native Calls**
  - This helps you **ditch JNI**, which is slow and painful.
  - Now you get:
  - Faster native calls
  - Zero-copy memory access
  - Unsafe-free native memory ops
- `MemorySegment segment = Arena.ofAuto().allocate(100);`
  - Perfect for:
  - High-performance networking
  - ML tensor operations
  - Databases
  - Custom encryption



## 5. Pattern Matching + Records + Sealed Classes = Compiler Optimizations

---

- 🧠 **11. Scoped Values (Preview)**
- Much faster than ThreadLocal for structured concurrency.
- Zero synchronization
- Compatibility with virtual threads
- Microsecond-level savings per access

# Quick Comparison: Java 17 vs Java 21

---

Area	Java 17	Java 21
Threads	OS threads	Virtual threads
GC	Stable	Faster, predictable
Startup	Good	Much faster
Pattern Matching	Limited	Complete & optimized
String ops	Good	Much improved
Native calls	JNI (slow)	FFM (fast)
Concurrency	Heavy	Lightweight

## Real-World Impact

---

- For microservices?
-  Faster startup
-  Lower latency
-  Lower memory
-  Handle more concurrent requests



## Real-World Impact

---

- For big-data / Kafka?
  - ⚡ Massive throughput
  - ⚡ Lower GC pause
  - ⚡ Faster serialization
- For banking apps (your use case)?
  - Faster crypto
  - Lower heap usage
  - Better thread-per-request model
  - Stable long-running performance

# New Tools and Utilities

---

-  **1. jcmd Enhancements**
  - jcmd got new diagnostic commands and richer output formatting.
  - Better JVM flag inspection
  - More detailed GC logs
  - Additional thread dump formats (virtual thread awareness)
  - Heap object statistics improvements
- Example:
- `jcmd <pid> GC.class_histogram`
- Now shows more accurate memory use.



## 2. jfr (Java Flight Recorder) Improvements

---

- JFR becomes even more helpful:
- New events for virtual-thread scheduling
- Better I/O profiling
- Reduced overhead for long-running monitoring
- New categories for FFM (Foreign Function & Memory API)
- Perfect for tuning microservices & high-performance apps.



## 3. jstat & jmap Improvements

---

- Better heap analysis
- Updated GC metrics
- Virtual thread awareness
- Native memory usage diagnostics refined



## 4. JDK Tooling Support for Virtual Threads

---

- Virtually all tools now understand virtual threads:
- jconsole
- jstack
- jcmt Thread.print
- Java Mission Control (JMC)
- Virtual threads behave like first-class citizens in monitoring tools.



## 5. Enhanced javac & jshell

---

- **javac:**
- Better pattern-matching error messages
- Record pattern support
- Exhaustive switch checking
- Improved incremental compilation performance
- **jshell:**
- Better REPL experience
- Added support for preview features
- Faster start time



## 6. Updated HTTP Client Diagnostics

---

- The newer built-in HTTP client (since Java 11) got:
- Faster debugging output
- Better SSL session debugging
- More detailed tracing with System properties



## 7. jlink / jpackage Enhancements

---

- Better custom runtime images
- Smaller image size
- Support for Windows/OSX/Linux platform-specific packaging improvements
- Better module resolution

## Security Enhancements & Updates

---

- Java 21 makes it official:
- The Security Manager is legacy and will be removed soon.
- Because modern applications run inside **containers**, not applets.



## 2. Enhanced Cryptographic Algorithms

---

- ★ **SHA-3 Performance Boosts**
- Hardware acceleration improved for:
- SHA3-256
- SHA3-512
- SHAKE algorithms



## 2. Enhanced Cryptographic Algorithms

---

- ★ AES/GCM Enhancements
- Faster AES/GCM using AVX-512
- Better performance with CPU AES-NI instructions
- Great news for banking apps, JWT signing, HTTPS, etc.



## 3. TLS & SSL Updates

---

- **TLS 1.3 improvements:**
- Faster handshake
- Better session reuse
- Lower latency
- **Stronger default TLS configurations:**
- Deprecated old cipher suites
- Removed weak RSA/DSA keys
- Enforced secure defaults in SSLEngine



## 4. PKI & Certificate Handling

---

- Stricter X.509 certificate validation
- Better OCSP stapling support
- Improved truststore formatting



## 5. Stronger Class-Loader Isolation

---

- Enhancements in module-system security:
- Fewer illegal-access loopholes
- Strong encapsulation of internal APIs
- Better warnings for reflective access



## 6. Foreign Function & Memory API Security

---

- FFM introduces:
  - Safer memory segments
  - Scoped lifetimes
  - No more “JNI memory leaks”
  - Meaning: you can safely interact with native code without blowing up your VM.

## 📌 7. Better Sandbox Enforcement for JVM Internals

---

Even though SecurityManager is going away:

Java 21 makes the JVM internal APIs harder to misuse:

- Restricted access to Unsafe
- Restricted access to internal sun.\* packages
- Better runtime checks



## 8. Strengthened Deserialization Filters

---

- Java tries to kill unsafe deserialization:
- Stronger default filters
- More enforcement around ObjectInputStream
- Reduced attack surface for RCE vulnerabilities



## 9. CVE Fixes & Hardening

---

- Java 21 bundles dozens of small but critical:
- Bug fixes
- Security patches
- Hardening updates

## Quick Summary Table

Feature	Summary
jcmd updates	Better GC, thread, heap diagnostics
jfr improvements	More events, virtual thread profiling
jstat/jmap	Enhanced heap & native memory analysis
javac upgrades	Better pattern matching, faster compile
jshell	Improved REPL with preview support
jlink/jpackage	Better packaging & smaller runtime images
HTTP Client tools	Better HTTP & SSL debugging
Tooling for finalization removal	Runtime warnings

# Security Enhancements

---

Feature	Summary
Security Manager removal	Deprecated for removal
SHA-3 hardware boost	Faster hashing
AES/GCM improvements	Faster encryption
TLS 1.3 improvements	Stronger & faster HTTPS
Stronger classloader rules	More encapsulation
FFM memory safety	Safer native code
Serialization filters	Default safer filters
CVE patches	Many small fixes

# Questions



# Module Summary

---

- Lambdas
- Stream API
- Base 64
- Nashorn
- Reflection
- Date Time API
- Repeated Annotations
- JDBC Improvements

