JVM Architecture, ClassLoading and Reflection

Read README.md ;) as well

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

Azat Satklichov
azats@seznam.cz,
http://sahet.net/htm/java.html

Agenda

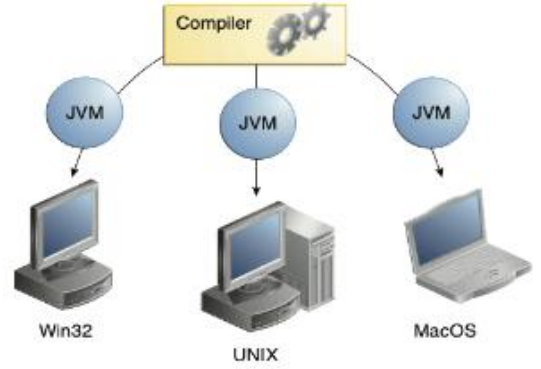
See also, demos:
"JVM Memory Management - Garbage Collection, GC Tools, ..
"Java Performance Tuning - JMH "
"Java Graal VM

- ❑ Java Platform
- ❑ Java Nostalgia - Sun Microsystems and Oracle
- ❑ JVM Architecture
- ❑ Class Loader Subsystem
- ❑ Built-in Class Loaders
- ❑ How do **C**lass **L**oaders work?
- ❑ Custom (User Defined) **C**lass **L**oaders
- ❑ Examples
- ❑ Reflection

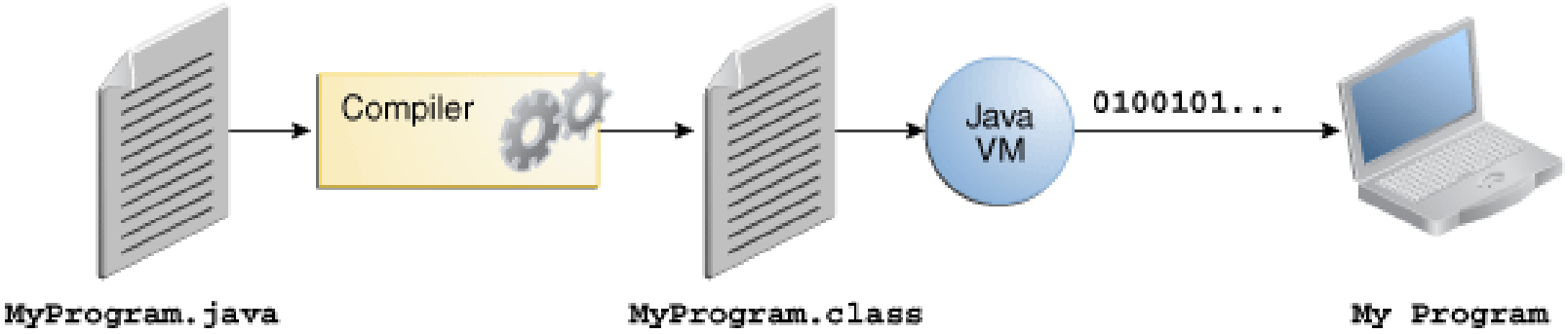
Java Platform

Five main goals which Java language intended to bring

- Simple, object-oriented, distributed
- Robust and secure.
- Architecture-neutral (agnostic) and portable.
- Execute with high performance.
- Interpreted, multi threaded, and dynamic.



Through the Java VM, the same application is capable of running on multiple platforms.



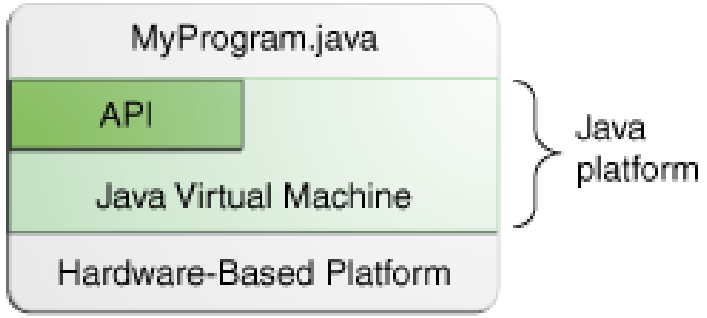
An overview of the software development process.

Java Platform

Java Platform

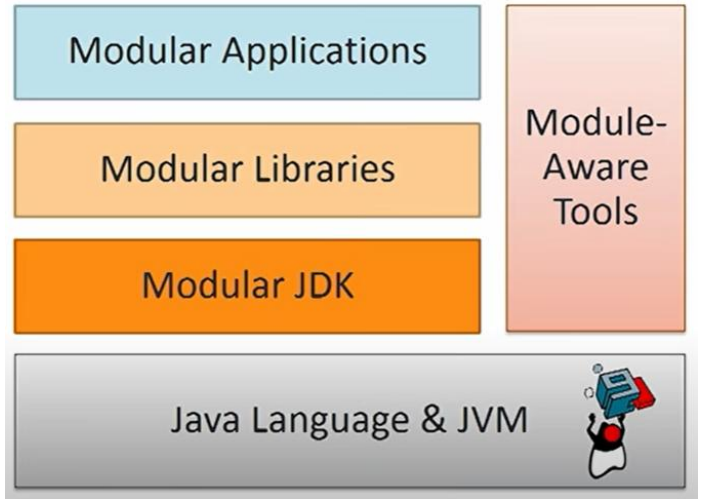
The Java platform is a software-only platform that runs on top of other hardware-based platforms (Win / Linux / MacOS).
The Java platform has two components:
- The Java Virtual Machine - The Java Application Programming Interface (API)

As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability. All Java Platforms consist of a JVM, an API, and other platform specific components.



The API and Java Virtual Machine insulate(isolated) the program from the underlying hardware.

Illustrates Java 8 and before.
Since Java 9, JRE is consumed in JDK. See sahet.net for Java Platform in detail



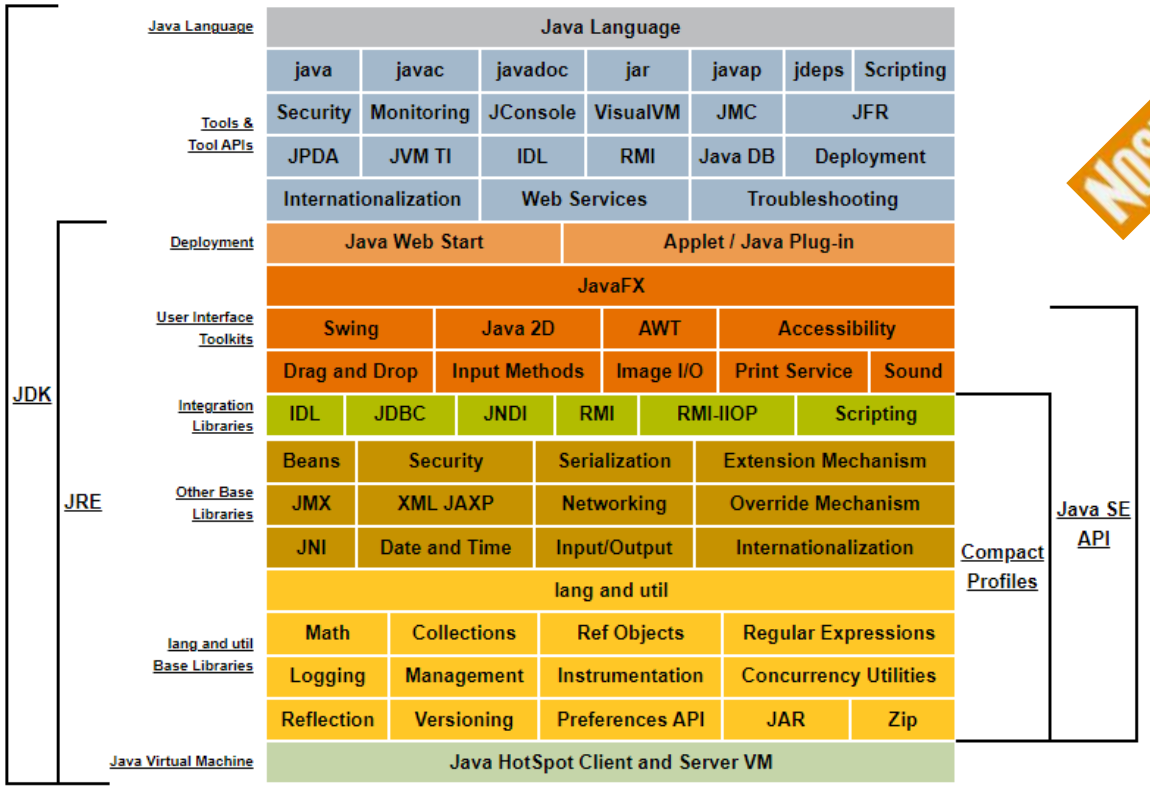
Java Platform since Java 9 - It is modular

Sun Microsystems and Oracle

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

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

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

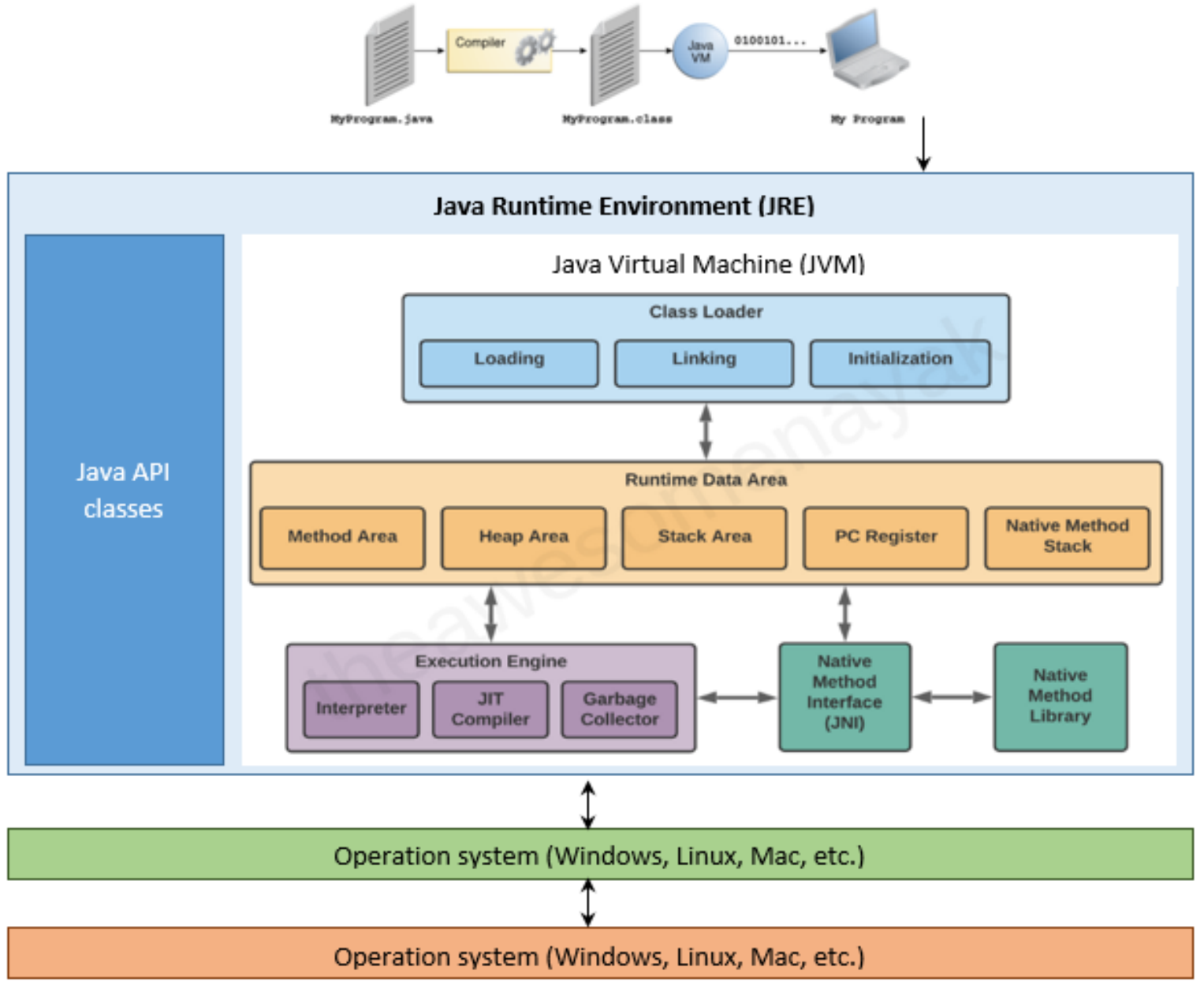


JVM Architecture

A virtual machine is a *virtual representation of a physical computer*. A **Java virtual machine** (**JVM**) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

The five major components inside JVM are

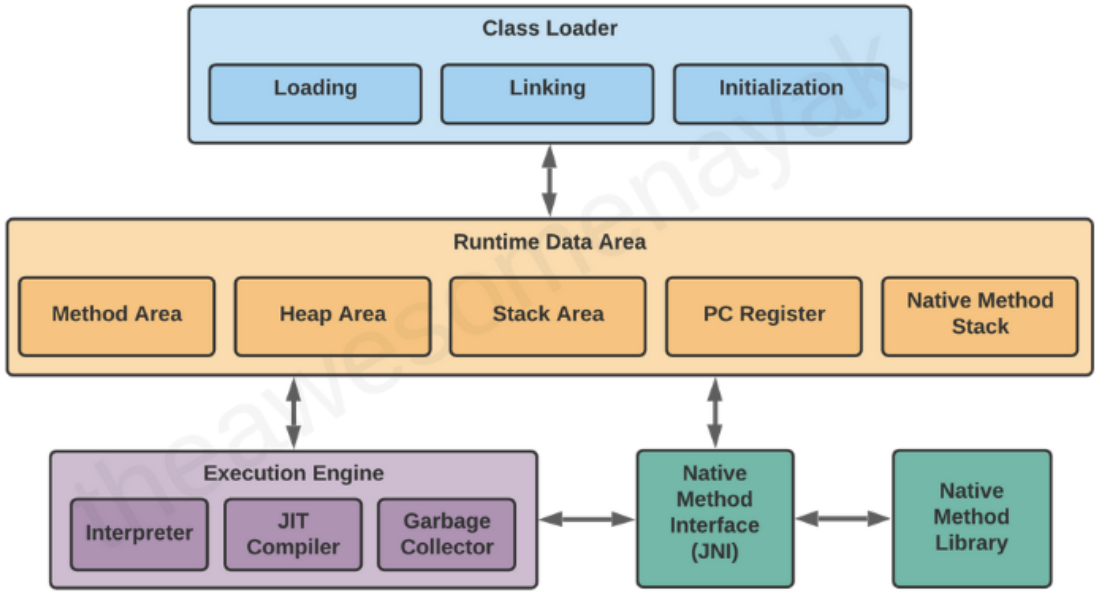
- Class Loader (loads class files to RAM)
- Memory Area (contains runtime data)
- Execution Engine (executes byte-code using interpreter)
- Native Method Interface
- Native Method Library



JVM Architecture

JVM Architecture

JVM (**consists of 3 components – Class Loader**, **Runtime memory** [data area], and **Execution Engine**) is only a specification, and its implementation is different from vendor to vendor – Oracle Hotspot, RedHat, Azul, etc. From Java 9 **"Extension Class Loader"** is replaced by **"Platform Class Loader"**



Class Loader subsystem - JVM runs on RAM, uses Class Loader subsystem to load all classes dynamically. It loads -> links -> initializes

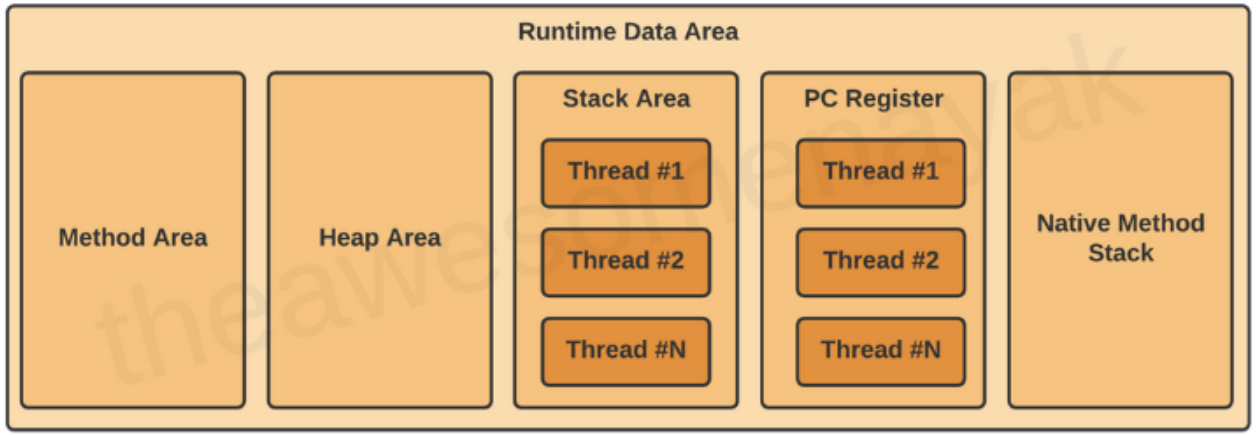
Runtime Data Area - Contains Runtime Data. Besides reading .class files ClassLoader Subsystem generates binary data info for each class and save it in Method Area

- Full qualified name of class and its direct parent
- Info about if class is Class/Interface/Enum
- Modifiers, static variables, and method info & etc.

Then for every loaded .class file, it creates one instance of Class to represent a file in Heap memory as in java.lang package. See demo: "JVM Memory Management - Garbage Collection, GC Tools, Java Ref"

Runtime Data Area

There are five components inside the runtime data area:



Method Area
All the class level data such as the run-time constant pool, field, and method data, and the code for methods and constructors, are stored here.

If the memory available in the method area is not sufficient for the program startup, the JVM throws an **OutOfMemoryError**

Heap Area - All the objects and their corresponding instance variables are stored here. This is the run-time data area from which memory for all class instances and arrays is allocated. JVM throws a **OutOfMemoryError**

Program Counter (PC) Registers
The JVM supports multiple threads at the same time. Each thread has its own PC Register to hold the address of the currently executing JVM instruction. Once the instruction is executed, the PC register is updated with the next instruction.

Native Method Stacks
The JVM contains stacks that support *native* methods. These methods are written in a language other than the Java, such as C and C++. For every new thread, a separate native method stack is also allocated.

Runtime Data Area

Stack Area – once new thread is created in the JVM, a separate runtime stack is also created at the same time. All local variables, method calls, and partial results are stored in the stack area. JVM throws a **StackOverflowError**

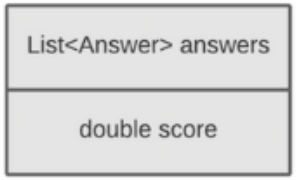
The Stack Frame is divided into three sub-parts:

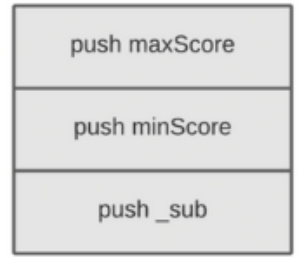
- **Local Variables –** Each frame contains an array of variables known as its *local variables*. All local variables and their values are stored here. The length of this array is determined at compile-time.
- **Operand Stack –** Each frame contains a last-in-first-out (LIFO) stack known as its *operand stack*. This acts as a runtime workspace to perform any intermediate operations. The maximum depth of this stack is determined at compile-time.
- **Frame Data –** All symbols corresponding to the method are stored here. This also stores the catch block information in case of exceptions.

For example assume that you have the following code:

In this code example, **variables** like **answers** and score are placed in the **Local Variables array**. The Operand Stack contains the variables and operators required to perform the mathematical calculations of subtraction and division.

```
double calculateNormalisedScore(List<Answer> answers) {
    double score = getScore(answers);
    return normalizeScore(score);
}
double normalizeScore(double score) {
   return (score - minScore) / (maxScore - minScore);
}
```





Note: Since the Stack Area is not shared, it is inherently thread safe.

Java Native Interface – JNI

This interface is used to interact with Native Method Libraries required for the execution and provide the capabilities of such Native Libraries (often written in C/C++). This enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.
At times, it is necessary to use native (non-Java) code (for example, C/C++). This can be in cases where we need to **interact with hardware, or to overcome the memory management and performance** constraints in Java. Java supports the execution of native code via the Java Native Interface (JNI).

You can use the native keyword to indicate that the method implementation will be provided by a native library. You will also need to invoke System.loadLibrary() to load the shared native library into memory, and make its functions available to Java.

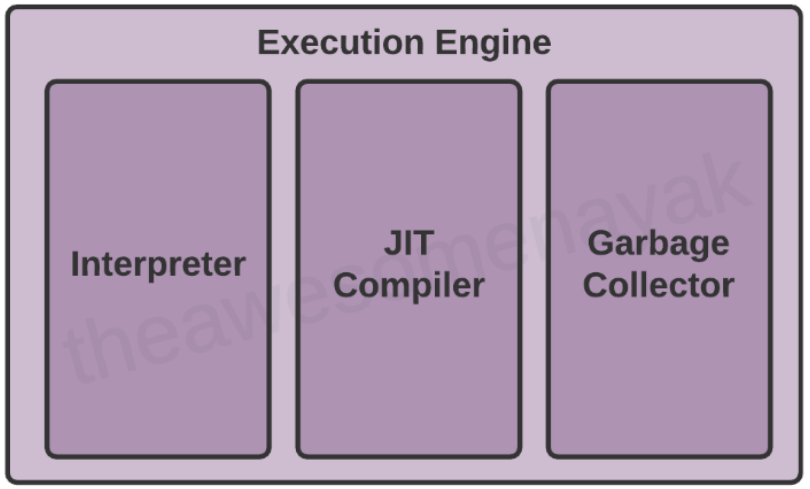
Native Method Libraries - Native Method Libraries are libraries that are written in other programming languages, such as C, C++, and assembly. These libraries are usually present in the form of **.dll** or **.so** files. These native libraries can be loaded through JNI.
This is a collection of C/C++ Native Libraries which is required for the Execution Engine and can be accessed through the provided Native Interface. . E.g. FinancialCalcAlgorithms written in C language

JVM Architecture

Execution Engine

Actual execution of byte-code happens here. Execution Engine executes the instructions in the bytecode line-by-line using interpreter by reading the data assigned to above runtime data areas.

Interpreter - before executing the program, the bytecode needs to be converted into machine language instructions. The JVM can use an **interpreter or a JIT compiler** for the execution engine. It interprets the bytecode and executes the instructions one-by-one. It can **interpret one bytecode line quickly**, but executing the interpreted **result is a slower task.**



JIT Compiler - The disadvantage of 'Interpreter' is that when one method is called multiple times, each time a new interpretation and a slower execution are required – this is **redundant operation**. JIT Compiler **fastens it** via **compiling all bytecode (.class) into native code (machine code)** then for repeated calls it provides native code and the execution using native code is much faster than interpreting instructions one by one. The native code is stored in the cache [**code cache** is non heap memory], thus the compiled code can be executed quicker.

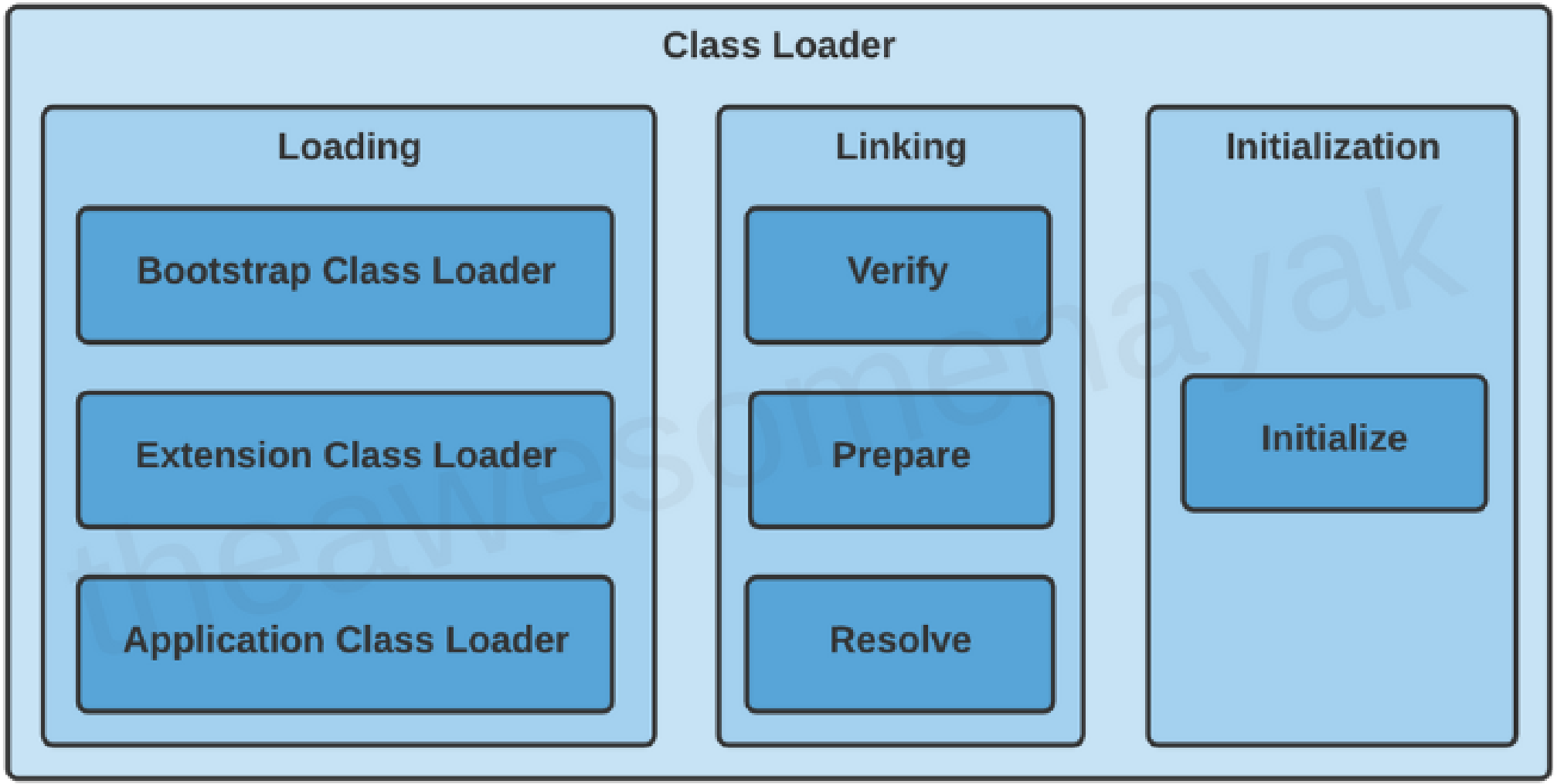
However, even for JIT compiler (C1, C2), it takes more time for compiling than for the interpreter to interpret. So look compiler optimizations, or GraalVM **AoT** Compiler .. See Java GralVM presentation for more detail

Garbage Collector - As long as an object is being referenced, the JVM considers it alive, one objects become eligible for GC JVM automatically removes them. See "JVM Memory Management - Garbage Collection, GC Tools, Java References"

Class Loader

loads -> links -> initializes

When you compile a **.java source file**, it is converted into byte code as a **.class file**. Once you run this program, the class loader loads this class into the main memory. First class to be loaded into memory is usually the class that contains the main() method. There are three phases in the class loading process: loading, linking, initialization



Class Loader Subsystem

Loading

Loading compiled classes (.class files) into memory is the major task of Class Loader. There are 3 types of class loaders [**AppClassLoader → PlatformClassLoader | ExtClassLoader → BootstrapClassLoader**] (connected with inheritance property) and they follow 4 major principles. 1. ***Visibility Principle*** 2. ***Uniqueness Principle 3. Delegation Hierarchy Principle 4. No Unloading Principle.*** The JVM uses the **ClassLoader.loadClass()** method for loading the class into memory. **NoClassDefFoundError or ClassNotFoundException** exception is thrown if class can't be loaded.

Linking

Linking involves in verifying and preparing a loaded class or interface, its direct super classes and super interfaces, and its element type as necessary. Linking occurs in 3 steps. [**Verification (if fail throws VerifyException)→ Preparation(allocate memory** for static storage and any data structures used by the JVM such as method tables**)** → **Resolution (**replace symbolic references from the type with direct references in runtime constant pool**)**

```
- consistent and correctly formatted symbol table
- final methods / classes not overridden
- methods respect access control keywords
- methods have correct number and type of parameters
- bytecode doesn't manipulate stack incorrectly
- variables are initialized before being read
- variables are a value of the correct type
```

Initialization (involves executing the initialization method of the class or interface (known as **<clinit>**))

Init. logic of each loaded class or interface **will be executed** (e.g. calling the constructor of a class) & make it **thread safe**). Init. is the final phase of class loading where all the static variables are assigned with their original values defined in the code and the static block will be executed (if any), and executed line by line from parent to child in class hierarchy.

Note: the JVM is multi-threaded. It can happen that multiple threads are trying to initialize the same class at the same time. This can lead to concurrency issues. You need to handle thread safety to ensure that the program works properly in a multi-threaded environment.

Built-in Class Loaders

During program execution compiled files (bytecodes - .class files) have to be loaded into the virtual machine. Class loaders are responsible for this activity. But where does Java load its classes from? From CLASSPATH

Setting the classpath globally – in Environment variable
Setting the classpath when running a Java application – using “-cp” option

Java classes aren't loaded into memory all at once, but rather when they're required by an application. This is where class loaders come into the picture. They're responsible for loading classes into memory.

Types of Built-in Class Loaders

```
public static void main(String[] args) throws ClassNotFoundException {
   System.out.println("Classloader of this class:" + A_Intro.class.getClassLoader());
   System.out.println("Classloader of DriverManager:" + DriverManager.class.getClassLoader());
   System.out.println("Classloader of ArrayList:" + ArrayList.class.getClassLoader());
}
```

Starting from Java9

```
Classloader of this class:jdk.internal.loader.ClassLoaders$AppClassLoader@c387f44
Classloader of DriverManager:jdk.internal.loader.ClassLoaders$PlatformClassLoader@36baf30c
Classloader of ArrayList:null //BootstrapClassLoader
```

Java 8 or before

```
Classloader of this class:jdk.internal.loader.ClassLoaders$AppClassLoader@c387f44
Classloader of DriverManager:jdk.internal.loader.ClassLoaders$ExtClassLoader@36baf30c
Classloader of ArrayList:null // //BootstrapClassLoader
```

Bootstrap Class Loader

Java classes are loaded by an instance of ***java.lang.ClassLoader***. However, class loaders are classes themselves. So the question is, who loads the ***java.lang.ClassLoader*** itself? It is **bootstrap or primordial** class loader.

It's mainly responsible for loading **JDK internal classes**, typically ***rt.jar*** ***(since Java 9 modules)*** and other core libraries located in the *$JAVA_HOME/jre/lib (/jmods)* directory. Additionally, the **Bootstrap class loader serves as the parent of all the other *ClassLoader* instances**.

This bootstrap class loader is part of the core JVM and is written in native code (C/C++). Different platforms might have different implementations of this particular class loader.

Extension (Platform) Class Loader

The extension class loader is a **child of the bootstrap class loader**, and takes care of loading the extensions of the standard core Java classes so that they're available to all applications running on the platform.
The extension class loader loads from the JDK extensions directory, usually the $JAVA_HOME/lib/ext directory, or any other directory mentioned in the java.ext.dirs system property.

Application or System Class Loader

It takes care of loading all the **application level classes** into the JVM. It loads files found in the classpath environment variable, -classpath, or -cp command line option. It's also a **child of the extensions class loader**. **And is written in Java.**

How do **C**lass **L**oaders work?

When the JVM requests a class, the **CL** tries to locate the class and load the class definition into the runtime using the fully qualified class name. The ***java.lang.ClassLoader.loadClass()*** method is responsible for loading the class definition into runtime. If the class isn't already loaded, it delegates the request to the parent **CL**. This process happens recursively. In case parent **CL** doesn't find the class, then the child class will call the java.net.URLClassLoader.findClass() method to look for classes in the file system itself. If the last child class loader can't find the class either, throws *java.lang.NoClassDefFoundError* or *java.lang.ClassNotFoundException*.

```
java.lang.ClassNotFoundException: abc.com.xyz.Demo at java.net.URLClassLoader.findClass(URLClassLoader.java:381) at
java.lang.ClassLoader.loadClass(ClassLoader.java:424) at java.lang.ClassLoader.loadClass(ClassLoader.java:357) at
java.lang.Class.forName0(Native Method) at java.lang.Class.forName(Class.java:348)
```

Delegation Model

Class loaders follow the delegation model, where on request to find a class or resource, a ClassLoader instance will delegate the search of the class or resource to the parent class loader. Only if the bootstrap and then the extension class loader are unsuccessful in loading the class, the system(application) class loader tries to load the class itself.

AppClassLoader → PlatformClassLoader|ExtClassLoader → BootstrapClassLoader

Unique Classes

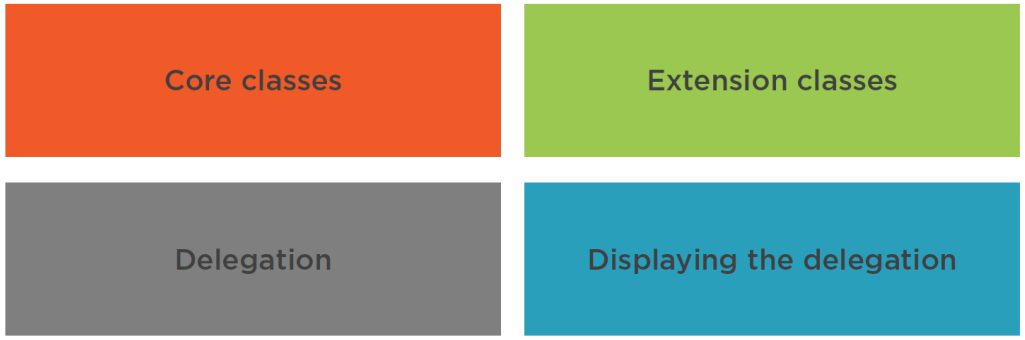
As a consequence of the delegation model, it's easy to ensure unique classes, as we always try to delegate upwards. If the parent class loader isn't able to find the class, only then will the current instance attempt to do so itself.

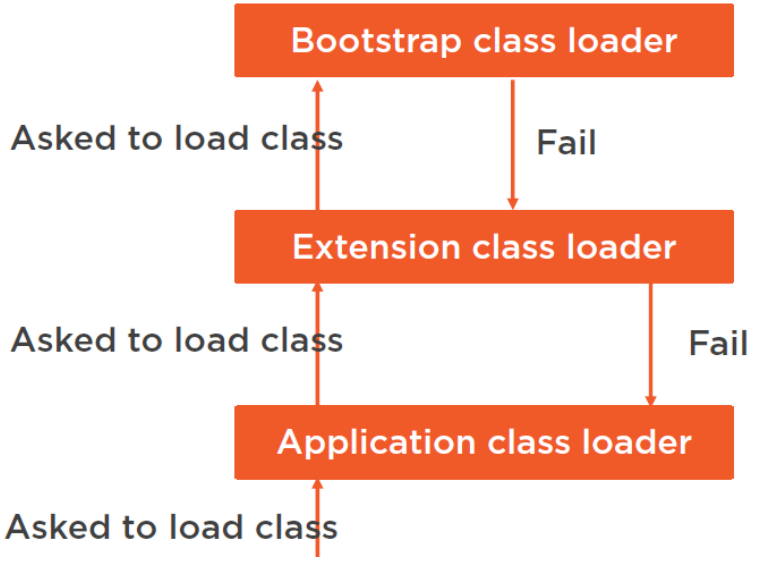
Visibility

Classes loaded by the system (application) class loader have visibility into classes loaded by the extension and bootstrap class loaders, but not vice-versa.

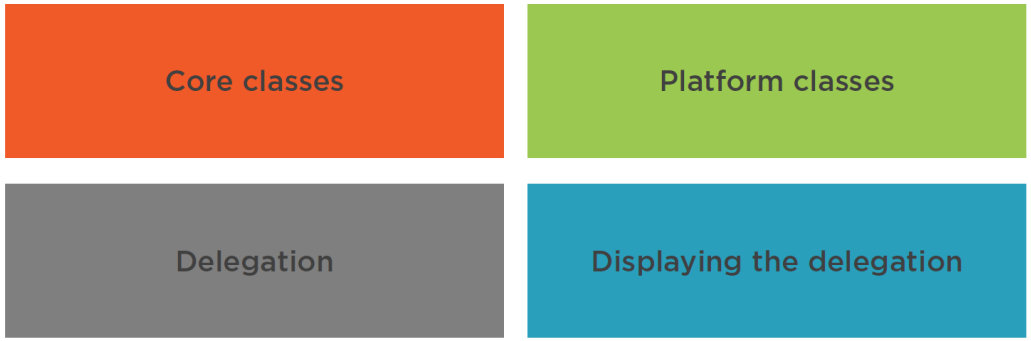
How do Class Loaders work?

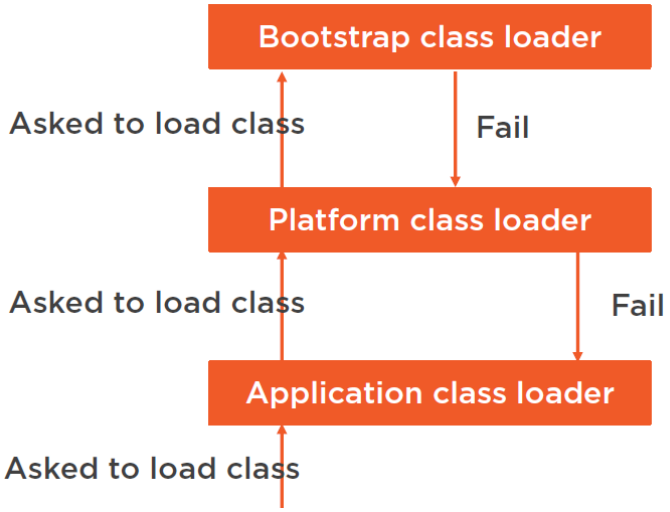
Class Loading Java 8 or prior





Class Loading Java 9 or later





Custom (User Defined) Class Loaders

*This guarantees the independence of applications through class loader delegation model. This approach is used in web application servers like Tomcat to make web apps and enterprise solutions run independently. Each Class Loader has its **namespace** that stores the loaded classes.*

1. Helping to modify the existing bytecode, e.g. weaving agents
2. Creating classes dynamically suited to the user's needs, e.g. in JDBC, switching between different driver implementations is done through dynamic class loading.
3. Implementing a class versioning mechanism [try to load same 2 class with different versions, e.g. it was done for Web. Apps via 2 different WAR files, because each WAR has own ClassLoaders] while loading different bytecodes for classes with the same names and packages. This can be done either through a URL class loader (load jars via URLs) or custom class loaders.

```
public class E_CustomClassLoader extends ClassLoader {

@Override
public Class findClass(String name) throws ClassNotFoundException {
byte[] b = loadClassFromFile(name);
return defineClass(name, b, 0, b.length);
}
private byte[] loadClassFromFile(String fileName) {
InputStream inputStream = getClass().getClassLoader()
.getResourceAsStream(fileName.replace('.', File.separatorChar) + ".class");
byte[] buffer;
ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
```

URLClassLoader typically used in your application
- Takes an array or URLs
- Can be file:///, htpp:///, jdbc:///, or some other protocol

Can write our own classloader
- Extend java.lang.ClassLoader
- May delegate first

Difference Between Class.getResource() and ClassLoader.getResource()

We can use the *getResource()* method on either a *Class* or *ClassLoader* instance to find a resource with the given name. The resource is considered to be data — for instance, images, text, audio, and so on. As a path separator, we should always use a slash ("/"). The method returns a URL object for reading the resource, or the *null* value if the resource cannot be found or the invoker doesn't have privileges to retrieve the resource.

Class.getResource()

We can pass an absolute or relative path when locating a resource with a *Class* object.

```
void givenAbsoluteResourcePath_whenGetResource_thenReturnResource() {
    URL resourceAbsolutePath = ClassGetResourceExample.class
        .getResource("/sahet/net/resource/test.txt"); //absolute path
//getResource("example.txt");//sing a relative path
    Assertions.assertNotNull(resourceAbsolutePath);
}
```

The process of finding the resource will be delegated to the class object's class loader. In other words,
the getResouce() method defined on the Class instance will eventually call ClassLoader's getResouce() method.

ClassLoader.getResource()

As the name suggests, the ClassLoader represents a class responsible for loading classes.
Every Class instance contains a reference to its ClassLoader.

To sum up, the differences between calling the getResource() method from the Class and ClassLoader instances. We can pass either a relative or an absolute resource path when calling the method using the Class instance, but we can only use the absolute path when calling the method on the ClassLoader.

Memory Issues

Once there is a critical memory issue, JVM throws exception

- **java.lang.StackOverFlowError** — thrown when Stack Memory is full
- **java.lang.OutOfMemoryError: Java heap space** thrown when Heap Memory is full
- **java.lang.OutOfMemoryError: GC Overhead limit exceeded** — thrown when app. Spend more time in GC
- **java.lang.OutOfMemoryError: Permgen space** — thrown when Permanent Generation space is full
- **java.lang.OutOfMemoryError: Metaspace** — thrown when Metaspace is full (since Java 8)
- **java.lang.OutOfMemoryError: Unable to create new native thread** — thrown when JVM native code can't create a new native thread in underlying OS because previously created threads consume all the available memory for the JVM
- **java.lang.OutOfMemoryError: request size bytes for reason** — when swap memory space is fully consumed by app.
- **java.lang.OutOfMemoryError: Requested array size exceeds VM limit**– app. uses an array size more than allowed in OS
- **StackOverlowError** – stack is ixed size, once exceeds its size this error is thrown

These exceptions can only indicate the impact that the JVM had, not the actual error. The actual error and its root cause conditions can occur somewhere in your code (e.g. memory leak, GC issue, synchronization problem), resource allocation, or maybe even hardware setting.

Therefore, you need to monitor resource usage, profile each category, go through heap dumps, check and debug/optimize your code etc. And if none of your efforts seems to work and/or your context knowledge indicates that you need more resources, go for it.

Common JVM Errors

Once there is a critical memory issue, JVM throws exception

ClassNotFoundExcecption - This occurs when the Class Loader is trying to load classes using Class.forName(), ClassLoader.loadClass() or ClassLoader.findSystemClass() but no definition for the class with the specified name is found.
NoClassDefFoundError - This occurs when a compiler has successfully compiled the class, but the Class Loader is not able to locate the class file at the runtime.
OutOfMemoryError - This occurs when the JVM cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.
StackOverflowError - This occurs if the JVM runs out of space while creating new stack frames while processing a thread.

THANK YOU

References

https://medium.com/platform-engineer/understanding-jvm-architecture-22c0ddf09722
https://www.baeldung.com/java-classloaders