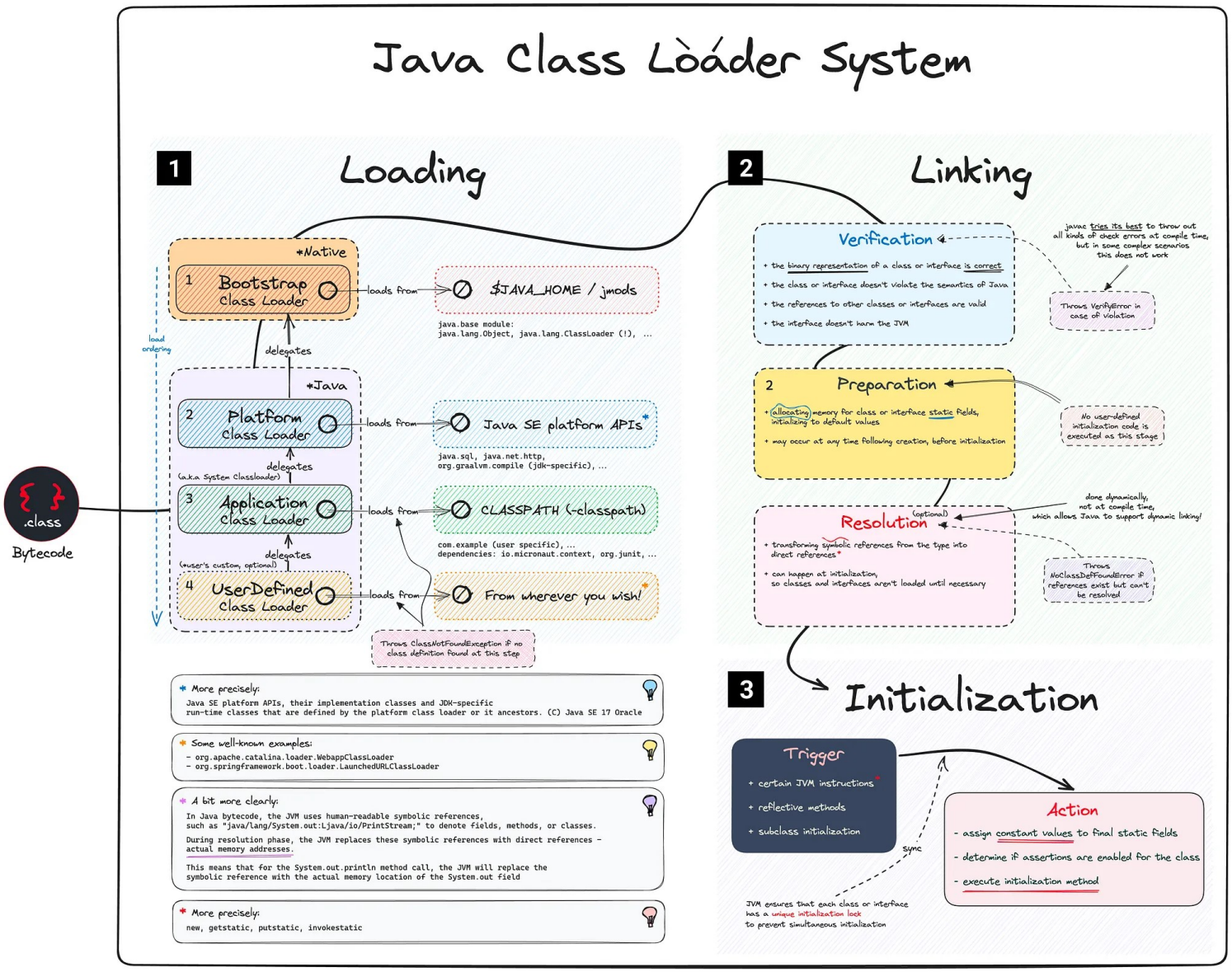
Java ClassLoader Internals

When I plunged into the world of Java class loaders, it was a response to a curious problem. Popular publications, supposed beacons of the Java world, are filled with conflicting and outdated information on the subject. This discrepancy sparked my investigation — a quest for clarity in the maze of Java class loaders.

As a Java developer, you have likely faced `ClassNotFoundException` or `NoClassDefFoundError` — cryptic messages that momentarily halt your coding flow. The online resources meant to shed light on these issues often add to the confusion instead.

Let's try to dive in together, stripping away the complexity. Here is a full picture of what we will explain:



Java Class Loader System, JDK 20 based

. . .

Proposition

Before we get further into the mechanics of class loaders, it's important to underscore one significant detail:

There is no "universal" Java Virtual Machine design.

The JVM is specified by Oracle Corporation, outlining what components and

behaviors are expected of any JVM. Yet, this specification doesn't dictate a single way of implementing these components. As a result, we find multiple, unique JVM implementations in the wild — such as HotSpot/OpenJDK, Eclipse OpenJ9, or a relatively hyped one, GraalVM (OpenJDK based). Each of these adheres to the JVM specification but may differ in various ways, including performance characteristics, garbage collection strategies, and, as you might guess, the details of class loading.

Another point to keep in mind is that:

> Java Virtual Machines are platform-dependent.

A JVM for a Windows OS isn't identical to a JVM for a Linux machine. "But wait," you might say, "I thought Java was all about write once, run anywhere — platform independence!" Absolutely correct. Java's platform independence doesn't mean the JVM is platform independent, though. It's quite the contrary.

Most articles on this topic also don't give a specific version of Java when describing it, which actually leads to misunderstanding, because the JVM evolves and changes with each version. It's autumn 2023, and the Java 21 just released, so we'll focus in it, relying on Oracle's JVM specification itself, and the Oracle Java SE documentation for simplicity.

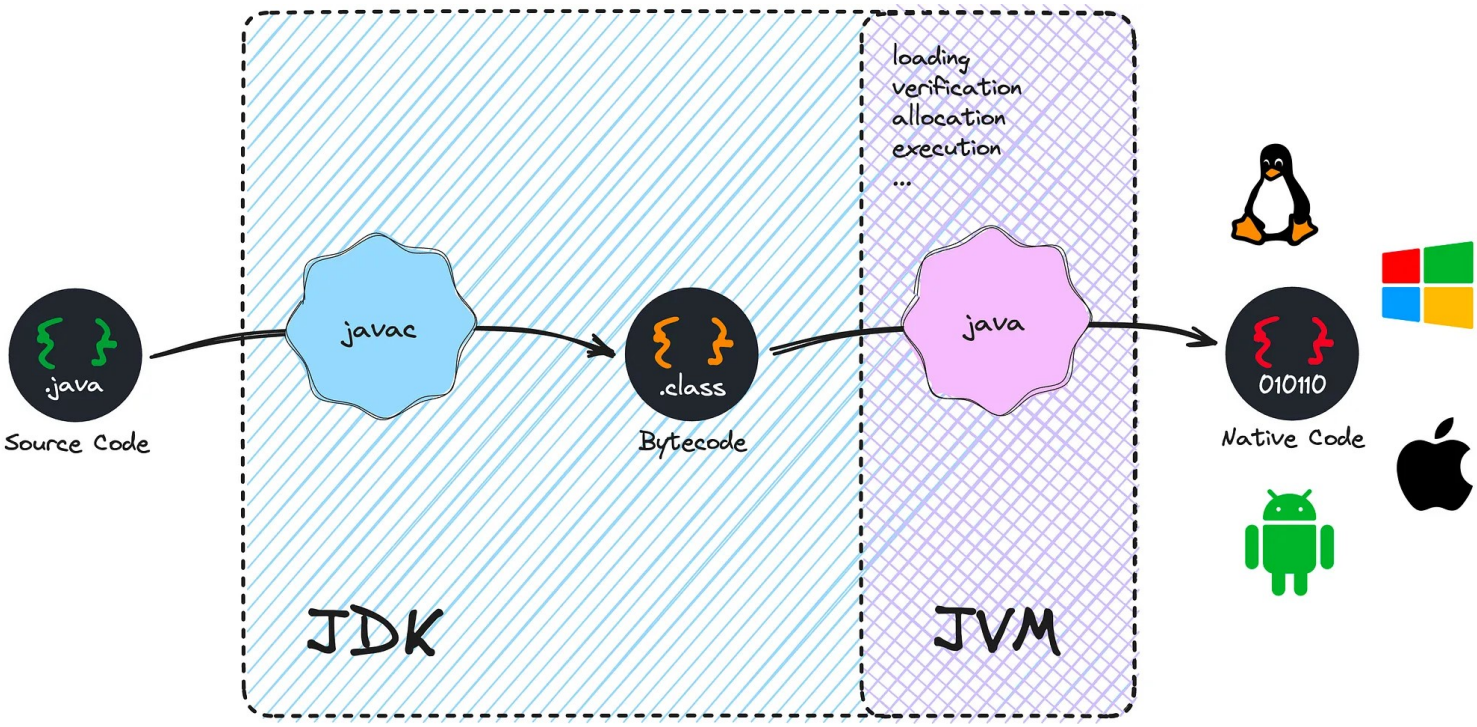
With this in mind, let's get back to our explo ration of the Java ClassLoader system.

. . .

Starting from the bottom

Simplified, when you run your application, the **JVM loads** the necessary classes into memory, **verifies** the bytecode, **allocates** necessary resources,

and finally **executes** the code by converting the bytecode into machine language instructions that the host machine understands.



Simplified Java source code to native flow

But what do *JVM loads* really mean? Java programs are comprised of classes and interfaces, written in human-readable Java code. To run this code on a machine, it needs to be translated into machine-understandable bytecode. This bytecode is stored in `.class` files, which the JVM can read and execute.

So when we talk about **"loading a class"**, we are referring to the process of **finding the appropriate .class file** with a particular name**, reading its contents, and bringing it into the JVM's runtime environment**, which is a specific portion of your machine's memory dedicated to running your application.

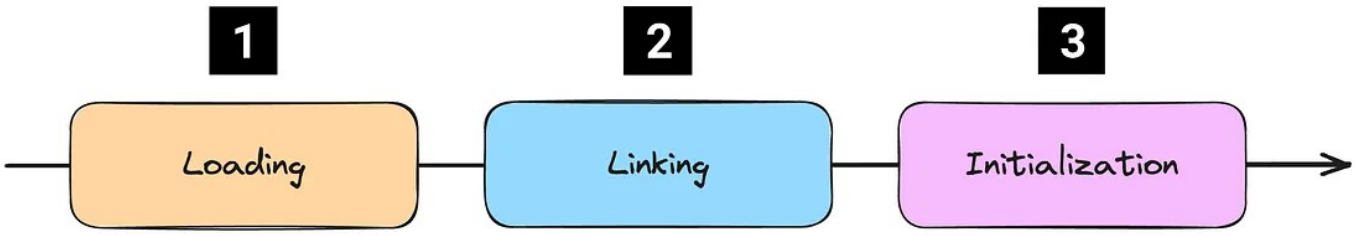
Or, if you prefer, a slightly more formal definition of "loading" from Oracle:

> *Loading* refers to the process of finding the binary form of a class or interface with a particular name, perhaps by computing it on the fly, but more typically by retrieving a binary representation previously computed

> from source code by a Java compiler, and constructing, from that binary form, a `Class` object to represent the class or interface.

Harden the explanation

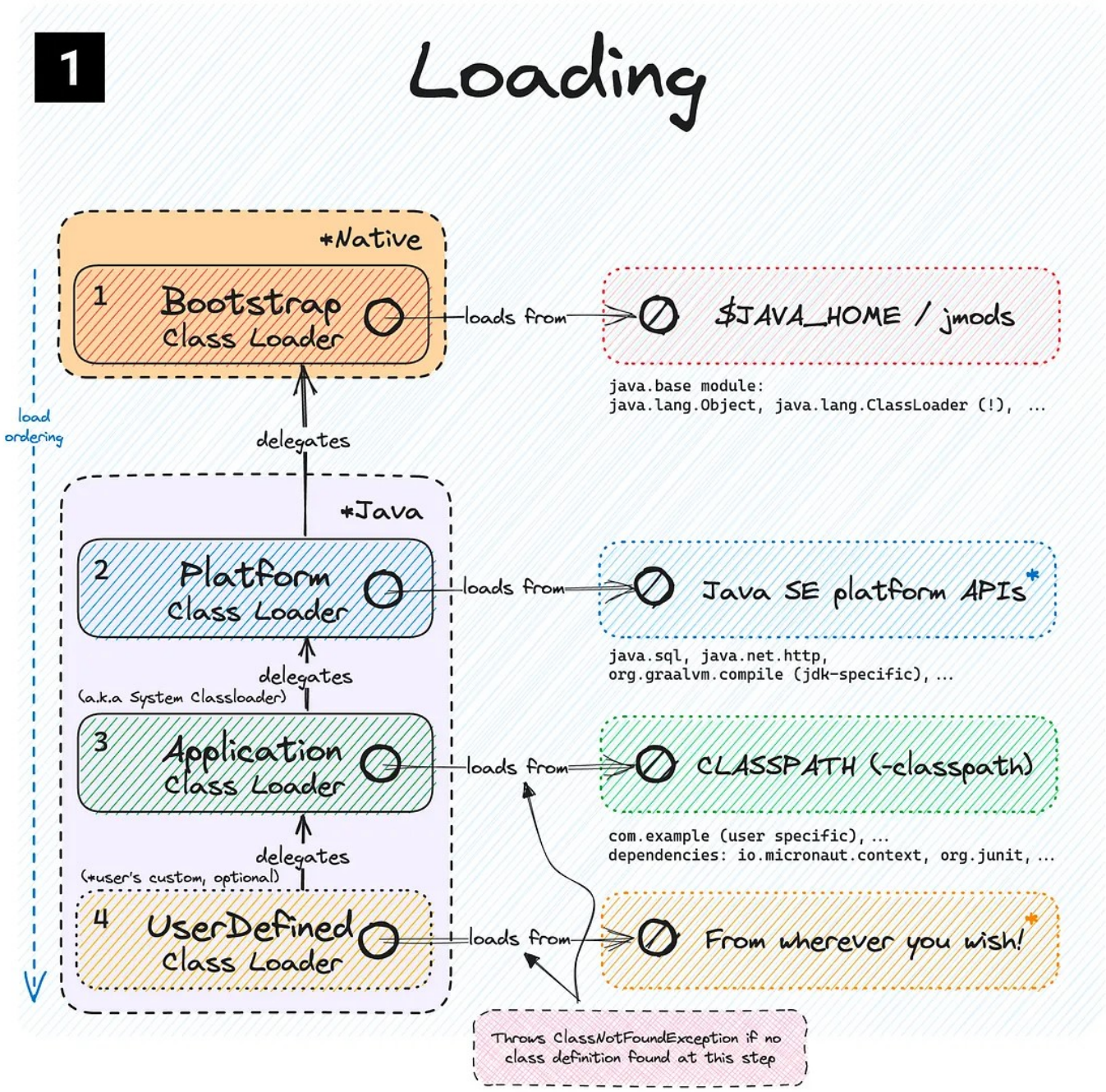
In reality, the ClassLoader System does more than just find the classes — it ensures the integrity and security of your Java application. by enforcing the Java runtime's binary structure and namespace rules. And it does this while providing the flexibility to load classes from various sources — not just the local file system, but also over a network, from a database, or even ones generated on the fly. Let's deep dive, breaking down the steps.



ClassLoader System Activities

1. Loading — The Initial Phase

The process begins when the ClassLoader is tasked with locating a specific class. This could be initiated by the JVM itself or triggered by a command in your code. Essentially, the ClassLoader's job is to take a fully qualified class name (like `java.lang.String`) and retrieve the corresponding class file (like `String.class`) from its location in storage into the JVM's memory.



Loading phase, OpenJDK 20 based

The Loading sub-system isn't a solo act; it's a hierarchical relay. Each ClassLoader, parent, and child, operate together, passing the baton of responsibility until the correct class is loaded.

The fundamental principles guiding this coordinated class loading process are:

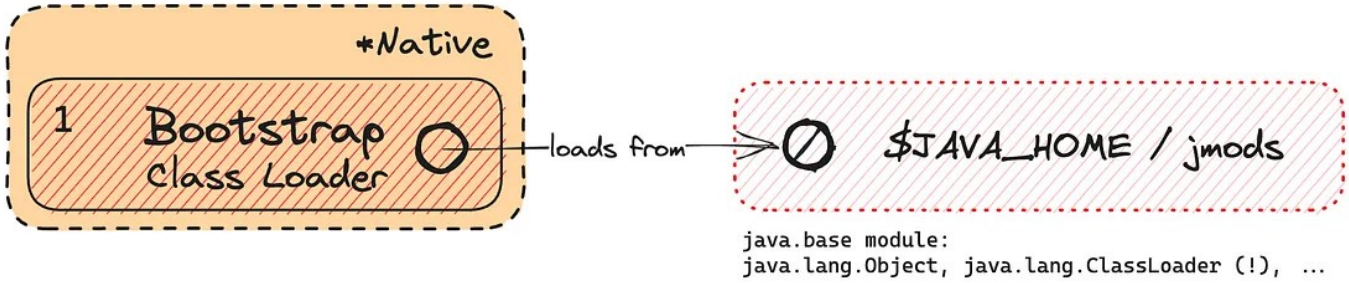
1. *Visibility*: a child ClassLoader can see classes loaded by its parent, **but not**

vice versa, ensuring encapsulation;

2. *Uniqueness*: **class loaded by a parent won't be loaded again by its child**, enhancing efficiency;
3. *Delegation Hierarchy*: the Application ClassLoader passes the class loading request up to the Platform and Bootstrap ClassLoaders. If they can't find the class, the request **cascades back down** the chain;

Let's deep dive into each ClassLoader now.

Bootstrap ClassLoader



Bootstrap ClassLoader

The oldest member of the family, the Bootstrap ClassLoader, is in charge of loading the core Java libraries located in the `<JAVA_HOME>/jmods` folder (such as *java.lang.*, *java.util.*, etc.) required by the JVM. Looking at the diagram, one can observe that other ClassLoaders are written in Java (*objects of java.lang.ClassLoader*), implying they also need to be loaded into the JVM — a task undertaken by the Bootstrap ClassLoader as well.

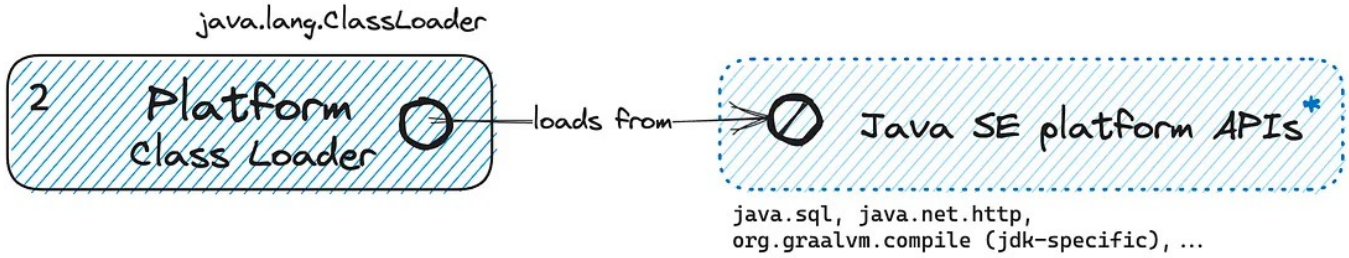
It's also worth noting that many resources describe the Bootstrap ClassLoader as the "parent" of the remaining classloaders. This signifies **a logical inheritance rather than direct Java inheritance** since Bootstrap ClassLoader **is written in *native* code**. This could be easily confirmed by the following line of code:

```
jshell> System.out.println(java.lang.ClassLoader.class.getClassLoader());
null
```

> *Bootstrap ClassLoader is also the only ClassLoader explicitly described in the Oracle specification. Definition of the remaining ones is called "User-defined" and left to the discretion of specific VM vendors.*

Platform ClassLoader

In my opinion, is the most controversial.



Platform ClassLoader

Java SE 20 documentation says the following:

> The platform class loader is responsible for loading the *platform classes*. Platform classes include Java SE platform APIs, their implementation classes and JDK-specific run-time classes that are defined by the platform class loader or its ancestors. The platform class loader can be used as the parent of a `ClassLoader` instance.

But what differentiates the *platform classes* from the *core classes* loaded by the Bootstrap ClassLoader? Let's attempt to observe what it essentially loads:

```
jshell> ClassLoader.getPlatformClassLoader().getDefinedPackages();
```

```
$1 ==> Package[0] { } // empty
```

It turns out that in an entirely empty Java program — absolutely nothing! Now, let's try to explicitly use a class from some standard package:

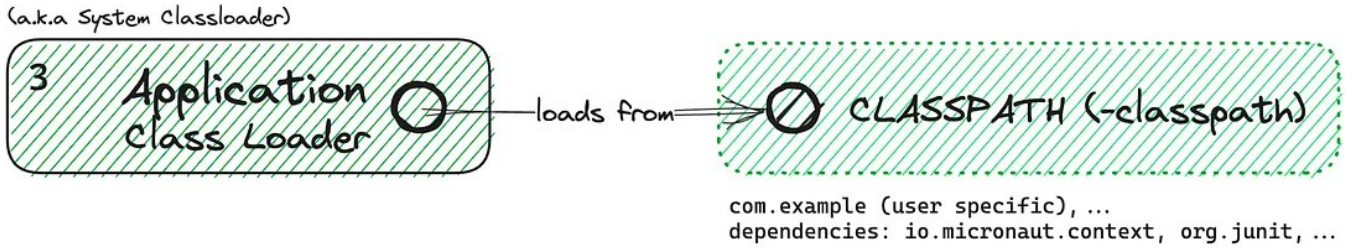
```
jshell> java.sql.Connection.class.getClassLoader()
$2 ==> jdk.internal.loader.ClassLoaders$PlatformClassLoader@27fa135a

jshell> ClassLoader.getPlatformClassLoader().getDefinedPackages()
$3 ==> Package[1] { package java.sql }
```

So to put it simply, Bootstrap loads the core runtime classes **required** to start the JVM, while the Platform loads the public types of the system modules, which a developer **might need**.

In this context, it is important to mention that many sources (such as Wikipedia, Baeldung) often refer to the Platform ClassLoader as the Extension ClassLoader. However, this isn't entirely accurate. It would be more correct to say that the Platform ClassLoader *has replaced* the Extension ClassLoader, which was used in Java 8 and earlier versions. This change came with the introduction of the Module System (JEP-261):

> *The extension class loader is no longer an instance of* `URLClassLoader` *but, rather, of an internal class. It no longer loads classes via the extension mechanism, which was removed by JEP 220. It does, however, define selected Java SE and JDK modules, about which more below. In its new role this loader is known as the* platform class loader*, it is available via the new* `ClassLoader::getPlatformClassLoader` *method, and it will be required by the Java SE Platform API Specification.*

Application ClassLoader



Application ClassLoader

Application ClassLoader, also known as System ClassLoader, is arguably the most commonly encountered in a day-to-day Java development environment. In Java SE 20, it still retains its traditional role and functionality.

This classloader is responsible for **loading all the classes from the classpath that has been set.** This could be from directories, JAR files, or other sources that have been specified in the classpath. It is here that most user-defined code is loaded when a Java application is launched.

```
public class MediumTeller {

    public static void main(String[] args) {
        // jdk.internal.loader.ClassLoaders$AppClassLoader@251a69d7
        System.out.print(MediumTeller.class.getClassLoader());
    }
}
```

From the standpoint of the classloader hierarchy, the Application ClassLoader is a *logical* child of the Platform ClassLoader. This means that when a class is loaded, and it's not found by the Application ClassLoader, the request is delegated upwards to the Platform ClassLoader and if necessary, further up to the Bootstrap, **ensuring a delegation mechanism.**

. . .

In addition to the three primary Class Loaders that we have discussed, you can also create your own User-Defined Class Loaders directly within their code. This feature provides an avenue for ensuring application independence, facilitated by the class loader delegation model. Web application servers such as Tomcat utilize this approach to ensure that different web applications and enterprise solutions can operate independently, even if they're hosted on the same server. We won't focus on that, since there are already enough guides on the topic of custom creation.

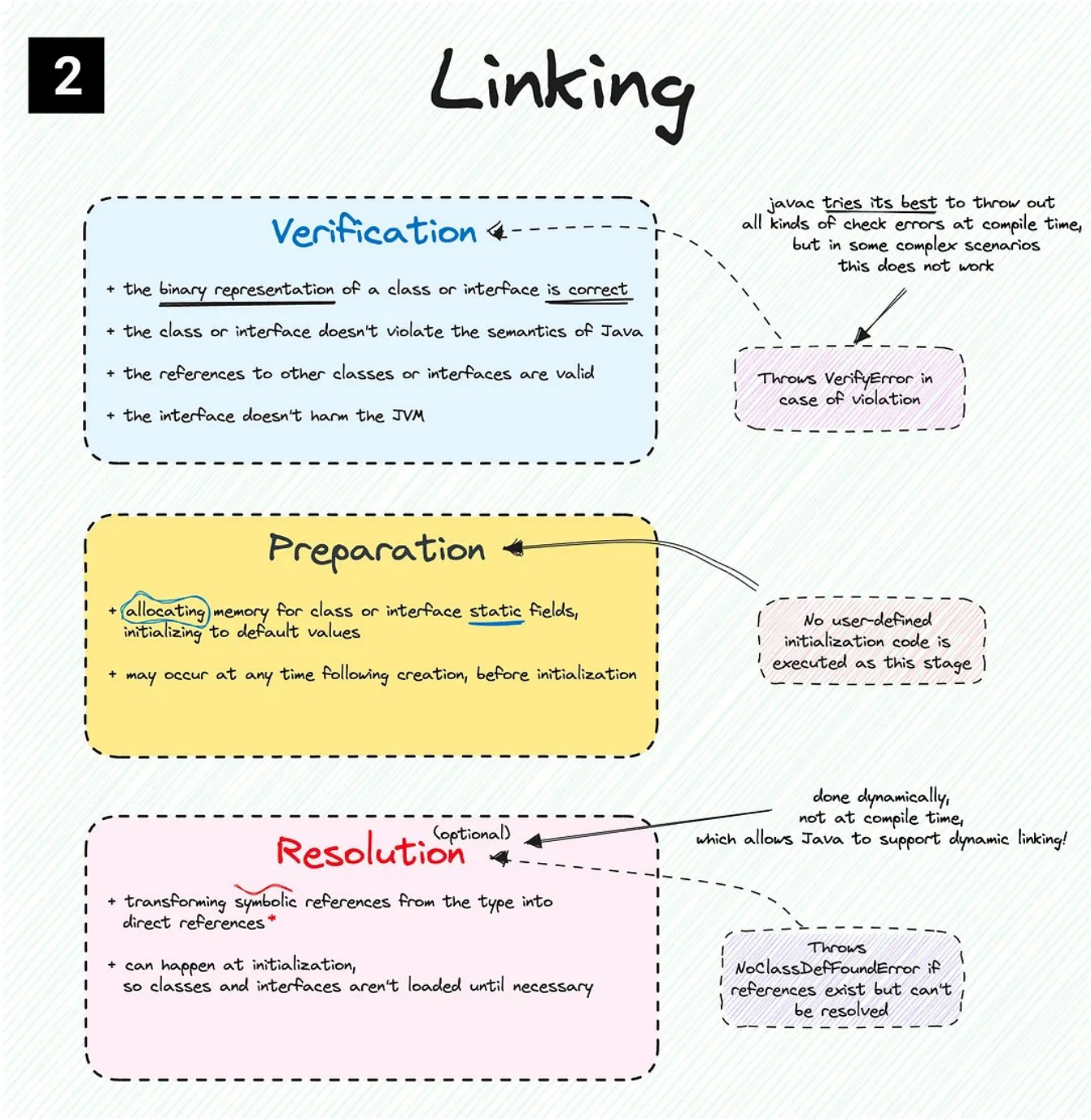
> Worth mentioning, that every Class Loader maintains its own namespace that records the classes it has loaded. When a ClassLoader is tasked with loading a class, it first consults this namespace, searching for the Fully Qualified Class Name (FQCN) to determine whether the class has already been loaded. Intriguingly, even when a class shares an identical FQCN with another if they exist in different namespaces, they are considered distinct classes. Having a class in a different namespace implies that it was loaded by a different ClassLoader, reinforcing the autonomy and separation between different parts of an application.

. . .

The outcome of the loading phase is a binary representation of a class or interface type in the JVM. However, at this point, the class is not yet prepared for usage.

2. Linking — Bridging the Gaps

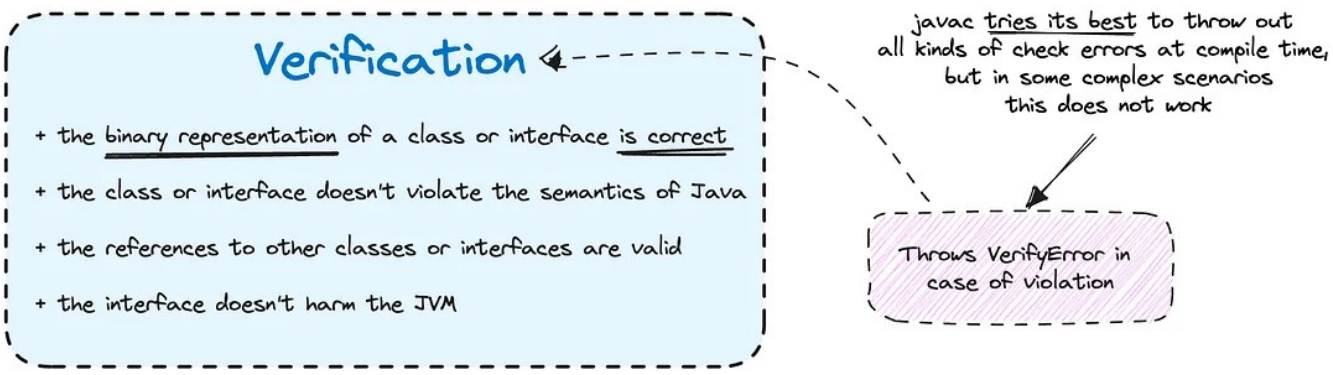
The linking phase involves several intricate steps to ensure the smooth execution of the program. This stage takes the loaded class or interface as input and performs essential tasks to verify the integrity of the code, prepare it for execution, and resolve any dependencies it might have.



Linking phase, OpenJDK 20 based

Once a class is loaded, it goes through a phase called linking. This phase involves a series of steps:

Verification



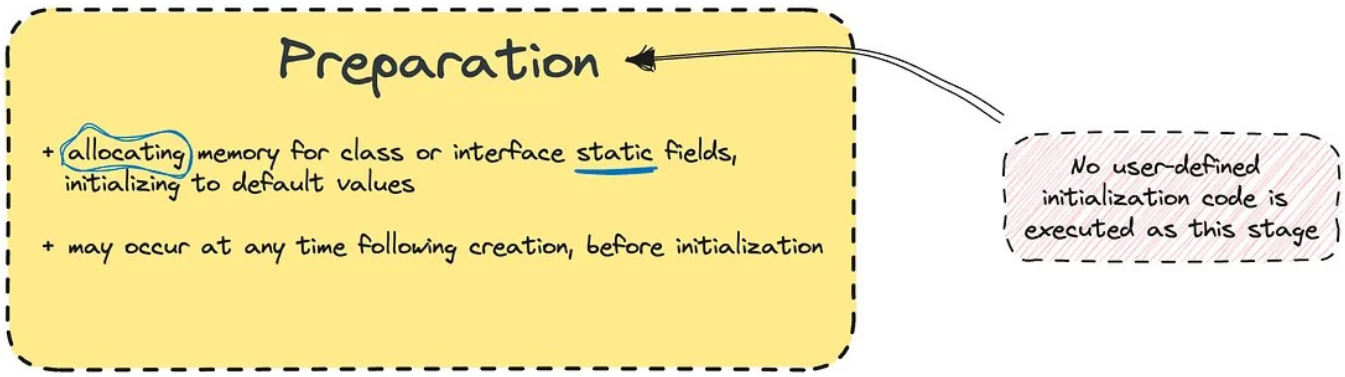
Verification phase

This stage is critical to maintaining the robustness of the Java runtime environment. It view the bytecode of the class or interface to **ensure its structural correctness, compatibility with the JVM**, and validation that it has been generated by a legitimate compiler.

In a world where Java programs can be transported over networks and potentially be generated by hostile compilers, this process becomes of paramount importance. It checks for consistency in the symbol table, whether final methods or classes have been overridden incorrectly, the correctness of access control keywords, the accurate number and type of parameters, correct stack manipulation, and more.

And finally, if the verification detects any anomalies, it throws a `java.lang.VerifyError`, leading to a `java.lang.LinkageError`.

Preparation



Preparation phase

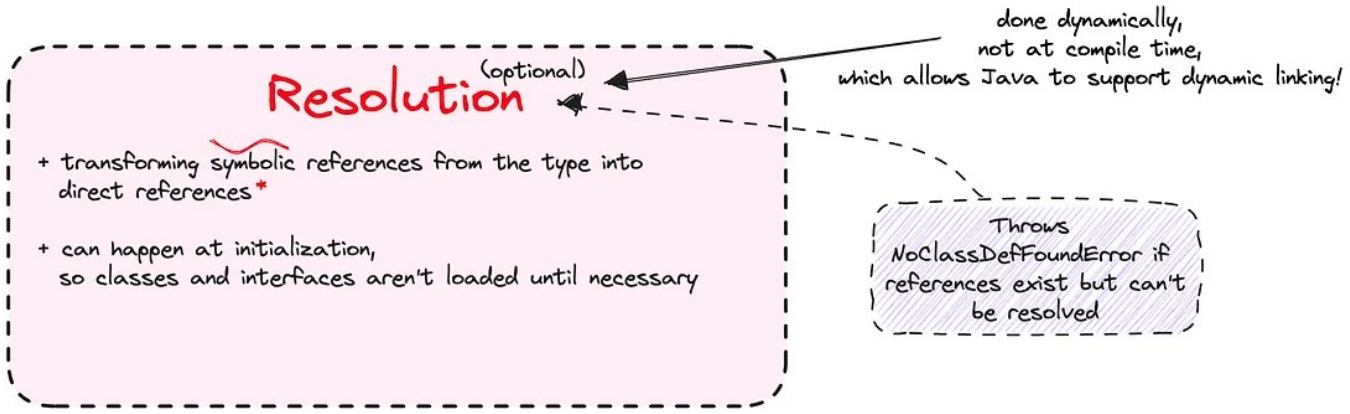
During this step, the JVM **allocates memory** for the class or interface's **static** variables and initializes them with their **default values.**

No user-defined initialization code is executed at this stage.

If the class or interface has instance fields, these too are allocated with memory and assigned default values, once the static fields of the entire class hierarchy are addressed. This preparation step allows for efficient runtime performance by laying the groundwork for the program's execution.

> Because linking involves the allocation of new data structures, it may fail with an `OutOfMemoryError`.

Resolution



Resolution phase

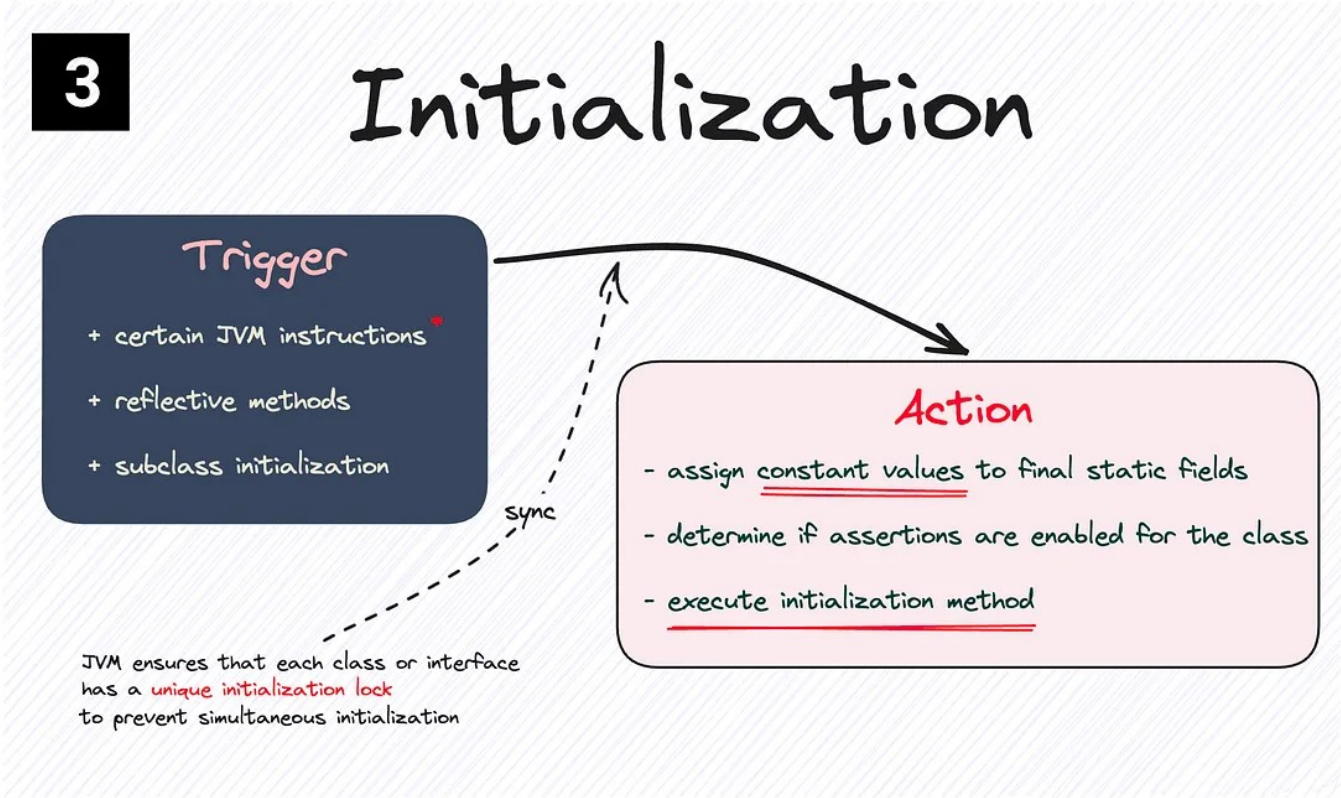
Here, any symbolic references in the class or interface — which are logical references that point to other classes or interfaces — are replaced with their actual memory locations. This transformation from symbolic references into direct references, often referred to as *Dynamic Linking*, **ensures that all dependencies of a class or interface are available at runtime.**

Interestingly, this step can be performed "lazily", that is, only when a

statement with a symbolic reference is executed. This approach, used by most JVMs, conserves resources as it prevents the unnecessary loading of classes or interfaces that may never be invoked.

> If a class referred by a symbolic reference cannot be located, a `java.lang.ClassDefNotFound` or `java.lang.ClassNotFound` exception is thrown.

3. Initialization



Initialization phase

Here, the initialization logic of each loaded class or interface will be executed (e.g. calling the constructor of a class). Since the JVM is multi-threaded, the initialization of a class or interface must be synchronized to prevent simultaneous initialization by multiple threads, **ensuring thread safety**.

The JVM invokes the special `<clinit>` method (the bytecode version of your static blocks and variable assignments), **setting all static variables to their**

designated initial values. At this point, the class is finally ready to be employed.

. . .

And there we go: application classes have been found, linked, initialized, and are now ready to be integrated into the bigger picture. The JVM now steps back, leaving the stage to your application. The classes, brimming with functionality and interconnected in an intricate web, are primed to breathe life into your application. Classes now can be created and manipulated, with methods invoked and variables set as defined by the logic of your app.

. . .

Appendix

- Oracle's Java SE API Documentation: java.lang.ClassLoader
- Oracle's JVM Specification: Chapter 5. Loading, Linking, and Initializing
- Oracle's Java Language Specification: Chapter 12. Execution
- Thilina Ashen Gamage: Understanding JVM Architecture