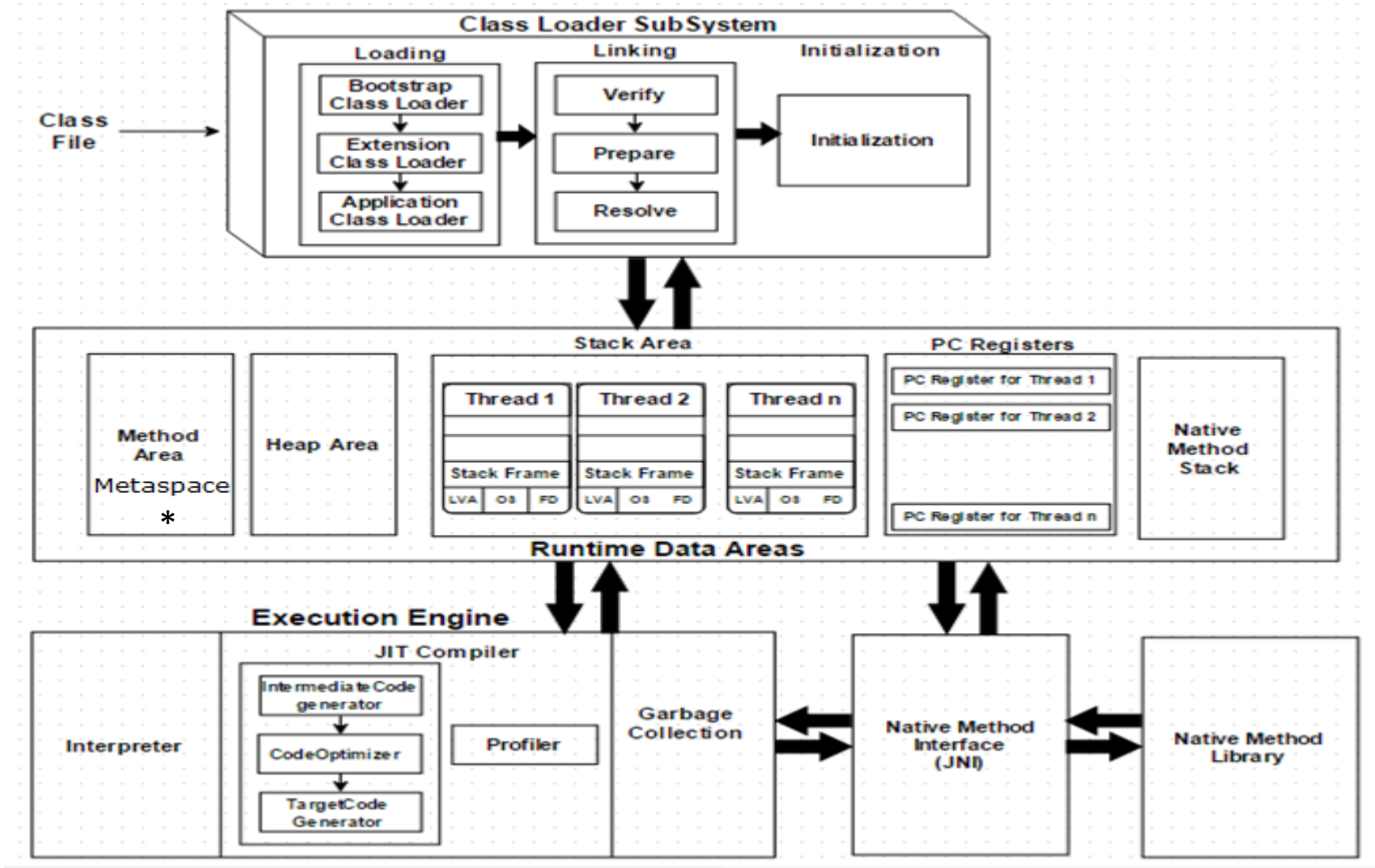
JVM Architecture Diagram



* Compressed Class Space

Class Loader Subsystem

Java's dynamic class loading functionality is handled by the class loader subsystem.

It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

Class Loading

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

Boot Strap ClassLoader – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.

Extension ClassLoader – Responsible for loading classes which are inside ext folder (jre\lib\ext).

-Djava.ext.dirs.=<directory-name>

* jdk 9 onwards, the extension class loader has renamed as a platform class loader.

System ClassLoader –Responsible for loading Application Level Classpath, path mentioned Environment Variable etc.

The above Class Loaders will follow Delegation Hierarchy Algorithm.

Note : Use OSGI/ Java Dynamic Modules

Linking

Verify – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.

Prepare – For all static variables memory will be allocated (Method area) and assigned with default values.

Resolve – Java class is compiled, all references to variables and methods are stored in the class's constant pool as a symbolic reference. All symbolic memory references are replaced with the original references from Method Area.

Initialization

This is the final phase of Class Loading, here all static variables will be assigned with the original values, and the static block will be executed.

Java Modules

A *Java module* is a packaging mechanism that enables you to package a Java application or Java API as a separate Java module.

A Java module is packaged as a *modular JAR file*.

A Java module can specify which of the Java packages it contains that should be visible to other Java modules which uses this module.

Before Java 9 and the Java Platform Module System you would have had to package all of the Java Platform APIs with your Java application.

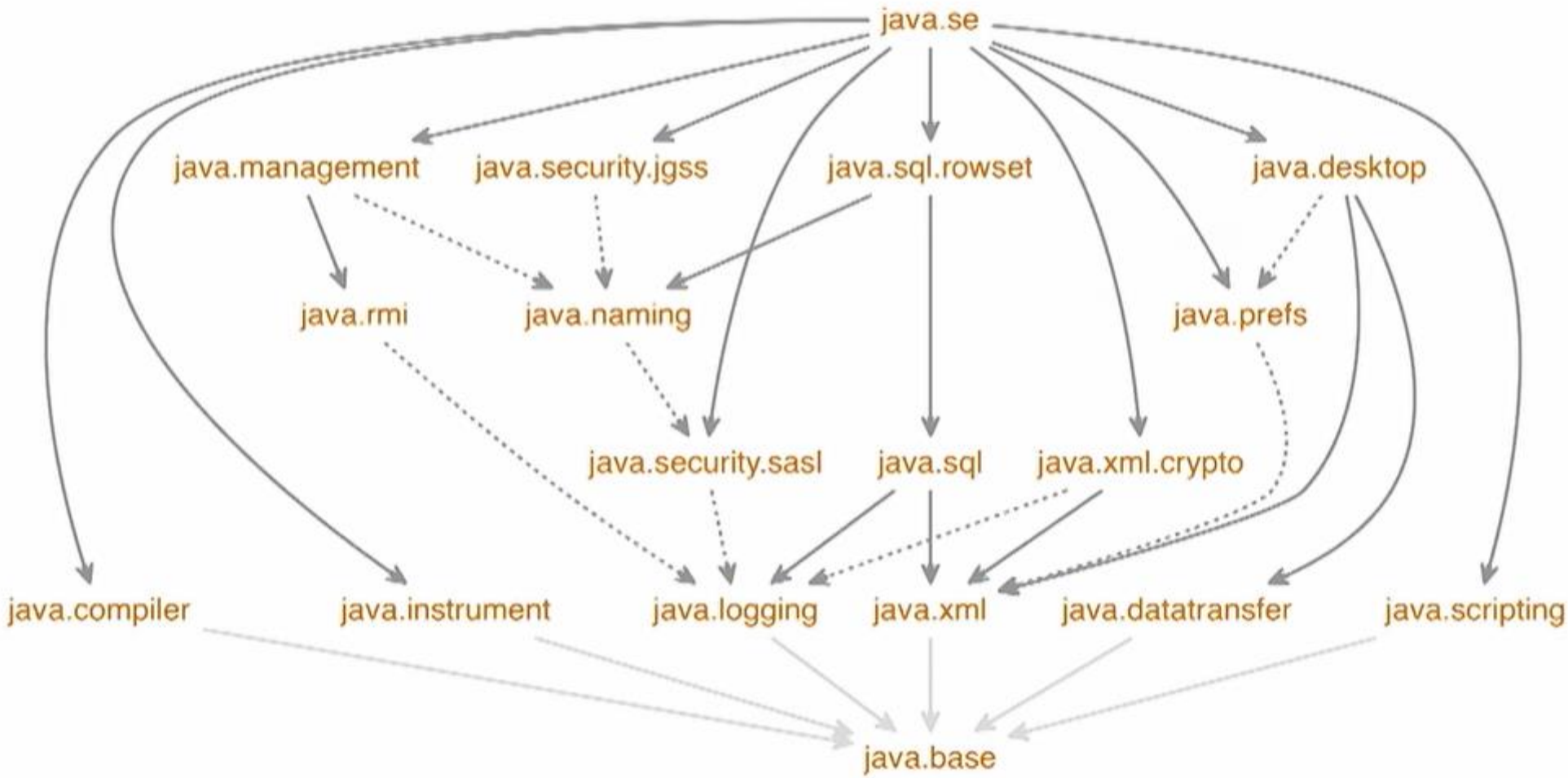
The unused classes makes your application distributable bigger than it needs to be.

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.

Java 9 restructured JDK into set of modules so that we can use only required module for our project.

The Modular JDK



Default Modules

When we install Java 9, we can see that the JDK now has a new structure. They have taken all the original packages and moved them into the new module system.

We can see what these modules are by typing into the command line:
java --list-modules

These modules are split into four major groups: java, javafx, jdk, and Oracle.

java modules are the implementation classes for the core SE Language Specification.

javafx modules are the FX UI libraries.

Anything needed by the JDK itself is kept in the jdk modules.

And finally, anything that is Oracle-specific is in the oracle module

Encapsulation of Internal Packages

A Java module must explicitly tell which Java packages inside the module are to be exported (visible) to other Java modules using the module.

A Java module can contain Java packages which are not exported. Classes in unexported packages cannot be used by other Java modules. Such packages can only be used internally in the Java module that contains them.

Packages that are not exported are also referred to as hidden packages, or encapsulated packages.

Modules Contain One or More Packages
A Java module is one or more Java packages that belong together. A module could be either a full Java application, a Java Platform API, or a third party API.

Module Naming
A Java module must be given a unique name.

com.examples.samplemodule

Module Root Directory
From Java 9 the Java Platform Module System offers an alternative directory structure which can make it easier to compile Java sources. From Java 9 a module can be nested under a root directory with the same name as the module.

com.examples.samplemodule/com/examples/samplemodule

Module Descriptor (module-info.java)

Each Java module needs a Java module descriptor named module-info.java which has to be located in the corresponding module root directory.

com.examples.samplemodule/com/examples/samplemodule/module-info.java

Module Exports

A Java module must explicitly export all packages in the module that are to be accessible for other modules using the module.

```
module com.examples.samplemodule {
    exports com.examples.samplemodule;
}
```

Module Requires

If a Java module requires another module to do its work, that other module must be specified in the module descriptor too. Here is an example of a Java module requires declaration:

```
module com.examples.samplemodule {
    requires javafx.graphics;
}
```