Exploring Java and Object-Oriented Programming

Features of OOP, Java's Evolution, and JVM Architecture

Object-Oriented Programming (OOP) Fundamentals

Encapsulation

Combining data and methods within a single unit, the object, promoting data security.

Inheritance

Creating new classes (subclasses) based on existing ones (superclasses), promoting code reusability and extensibility.

Polymorphism

The ability of an object to take on multiple forms, allowing for flexible and adaptable code.

Abstraction

Hiding complex details and presenting a simplified interface, promoting code readability and maintainability.

Java's Evolution: A Timeline

1

1991

James Gosling and his team at Sun Microsystems start the development of **Oak**, which later evolves into **Java**.

2

1995

Java 1.0 is released, marking the official birth of the language. Java quickly gains popularity for its portability and security.

3

1998

Java 2 is released, introducing significant improvements, including Swing GUI toolkit and J2SE, J2EE, J2ME.

4

2004

Java 5 is released, introducing generics, autoboxing, and enhanced for loop, making Java more powerful and developer-friendly.

5

2014

Java 8 is released, bringing Lambda expressions, stream API, and date/time API, revolutionizing Java's coding style and efficiency.

Introducing Java

High-Level Language

Easy to read and write, abstracting away low-level details.

Object-Oriented

Built upon the principles of encapsulation, inheritance, and polymorphism.

Platform-Independent

Runs on any platform with a Java Virtual Machine (JVM).

WORA

Code compiled once can run on various operating systems without modification.

Strongly Typed

Data types are strictly enforced, reducing runtime errors.

Robust

Exception handling mechanisms enhance reliability.

Automatic Memory Management

Garbage collection preventing memory leaks.

Versatile

Java is used for web apps, mobile apps, enterprise software, and more.

Security and Platform Independence

1 **Secure Code Execution**

Java's bytecode verification and security manager prevent malicious code from running, ensuring a secure execution environment.

2 Platform Independence

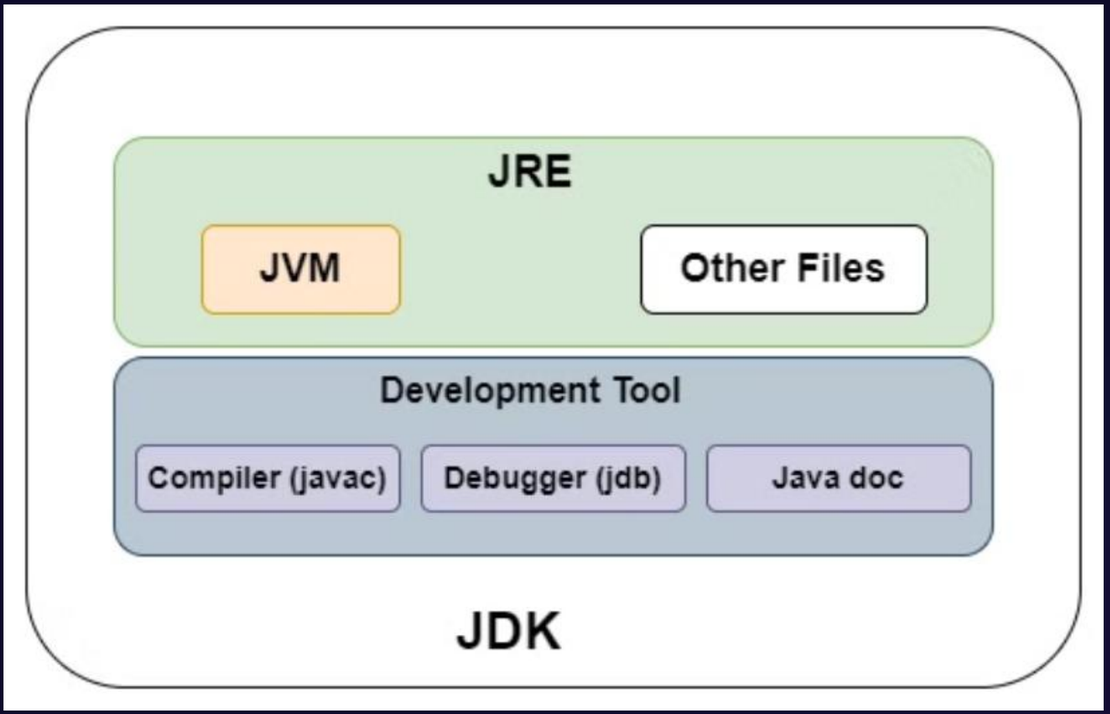
- **WORA (Write Once, Run Anywhere)**: Java code is compiled into **bytecode** that is interpreted by the JVM, making Java applications platform-independent.
- The JVM abstracts the underlying OS, so Java programs can run on any device with a JVM installed.

JVM, JRE & JDK

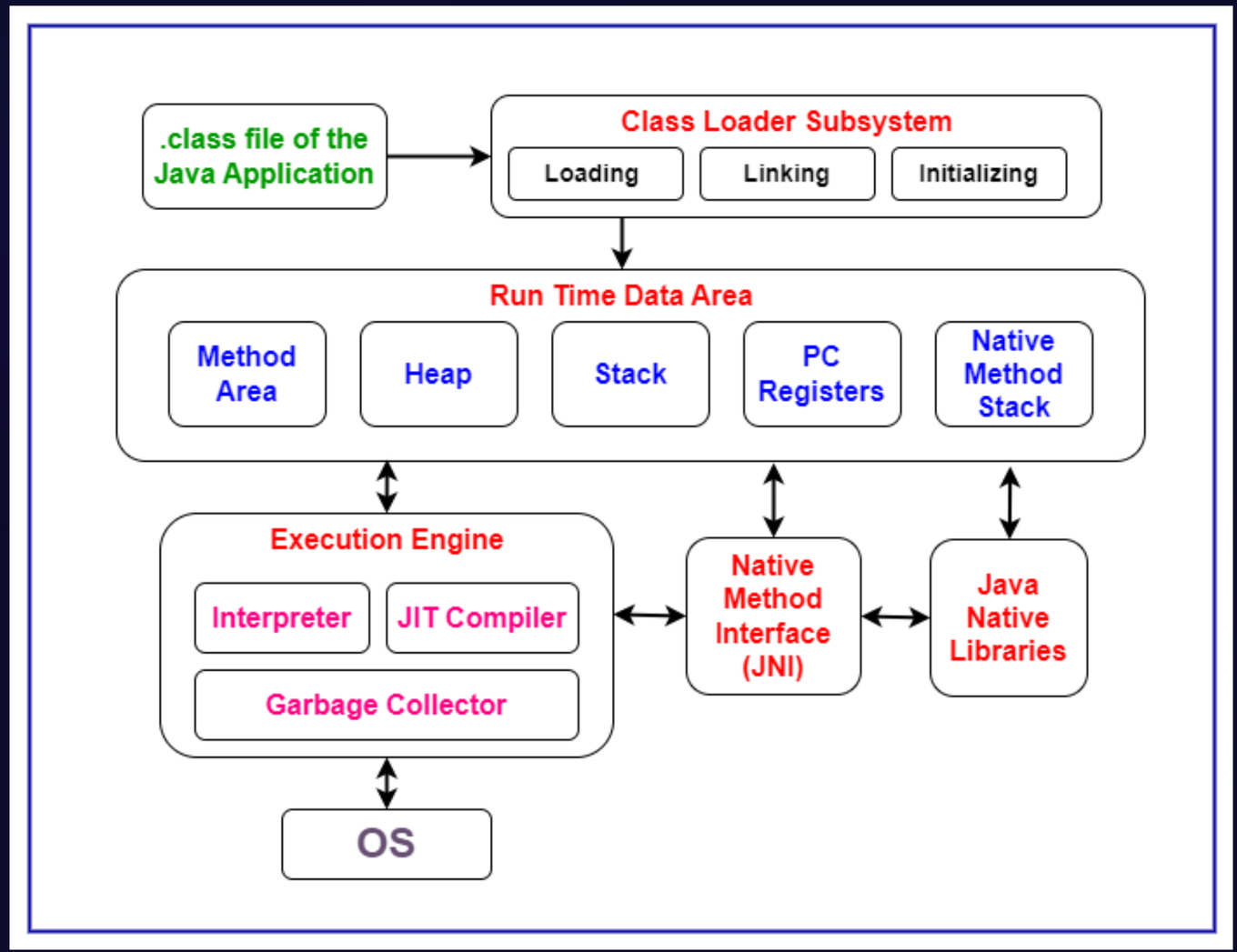
The Java Virtual Machine (JVM) is the runtime environment that executes Java bytecode.

The **Java Runtime Environment (JRE)** includes the JVM plus libraries and other components needed to run Java applications.

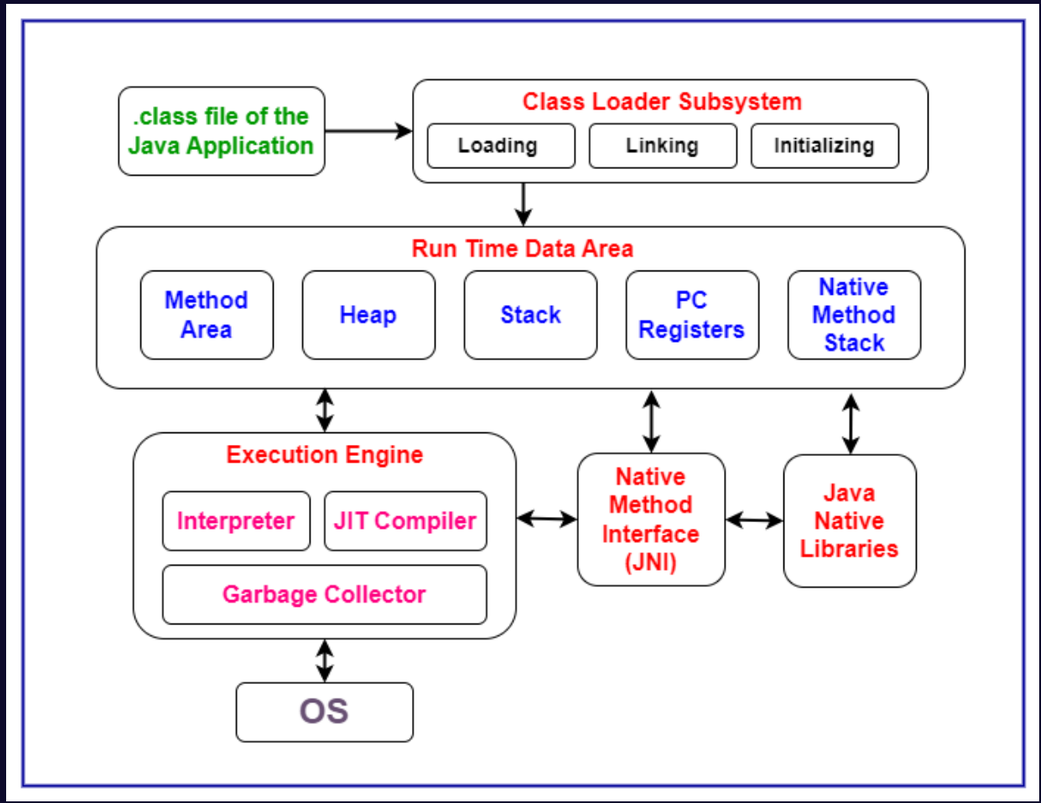
The **Java Development Kit (JDK)** contains the JRE and additional tools for developing Java applications, such as the compiler and debugger.



Java Virtual Machine (JVM) Architecture



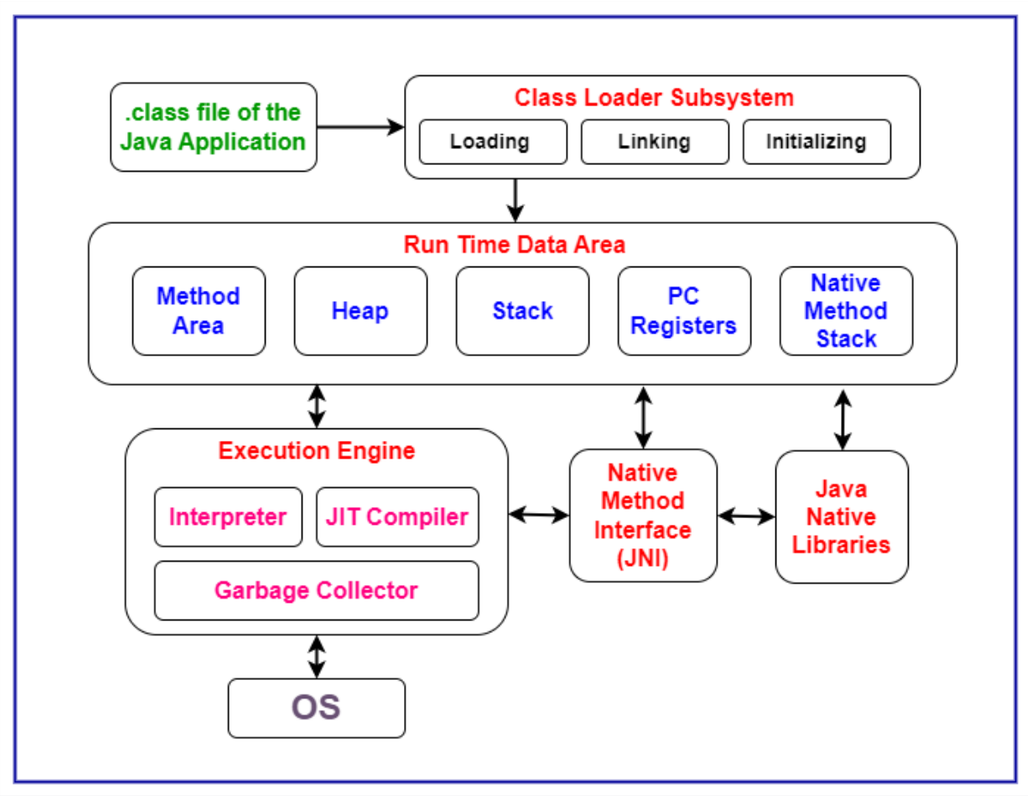
Java Virtual Machine (JVM) Architecture



The **JVM Class Loader Subsystem** is responsible for loading, linking, and initializing classes. It loads a class file into RAM during execution.

- ✓ **Loading** involves finding and importing class files into the JVM.
- ✓ **Linking** verifies the class and its super class, prepares memory, and resolves symbolic references.
- ✓ **Initialization** Loaded class is initialized by calling constructor and allocates static variables.

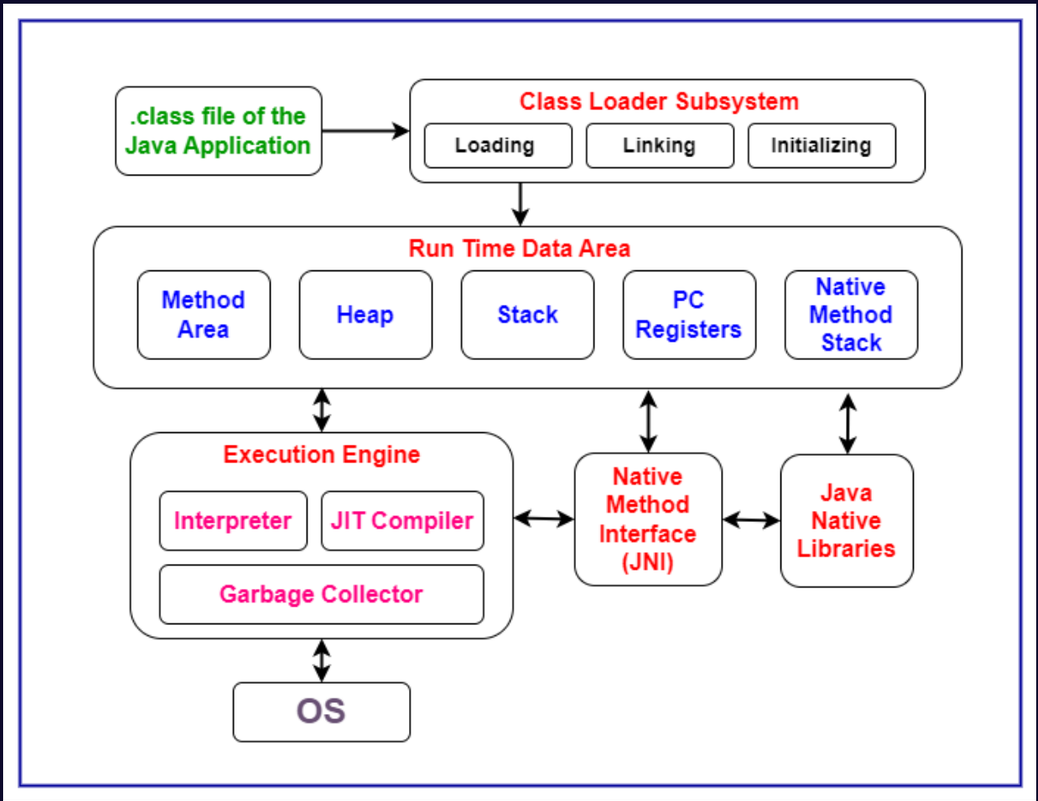
Java Virtual Machine (JVM) Architecture



The JVM runtime data area consists of several key components.

- ✓ **Method Area**: Stores class-level information (metadata, static variables).
- ✓ **Heap**: Stores objects created during runtime.
- ✓ **Stack**: Stores method calls, local variables, and references.
- ✓ **PC Register**: Keeps track of the instruction being executed.
- ✓ **Native Method Stack**: Handles native method calls.

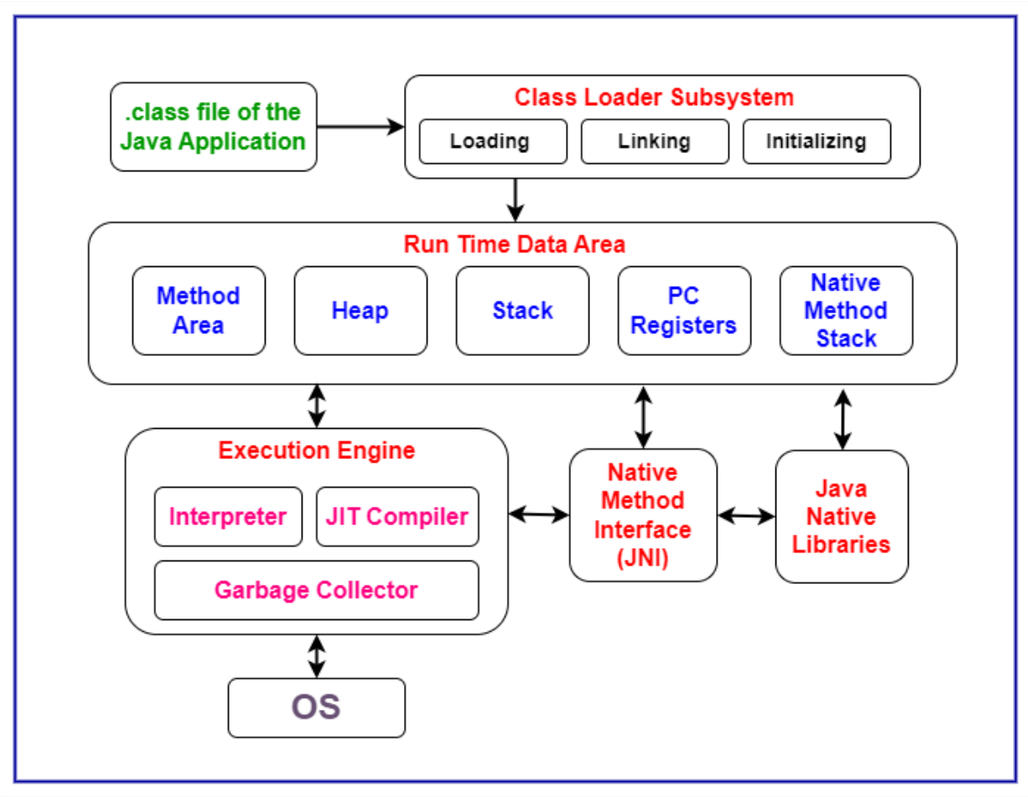
Java Virtual Machine (JVM) Architecture



The JVM Execution Engine is responsible for executing Java bytecode. It comprises several key components:

- ✓ **Interpreter:** Executes bytecode instructions one by one.
- ✓ **Just-In-Time (JIT) Compiler:** Code optimization, complies frequently executed bytecode segment into native machine code and cached for faster execution.
- ✓ **Garbage Collector:** Automatically manages memory by reclaiming unused objects. This prevents memory leaks and simplifies memory management for developers.

Java Virtual Machine (JVM) Architecture



The **Java Native Interface (JNI)** allows Java code to interact with native libraries (code written in other languages like C or C++). This is crucial for accessing system-specific resources or integrating with legacy systems.

A **Java Native Library (JNL)** is a shared library containing native code that is accessed via JNI. JNI enables extensibility by allowing Java to leverage code from outside the JVM.

Thank You