**JAVA**

The Oracle implementation is packaged into two different distributions:

1. **Java Runtime Environment (JRE)** which contains the parts of the Java SE platform required to run Java programs and is intended for end users.
2. **Java Development Kit (JDK)** which is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.

A diagram of a computer program

Description automatically generated

**JDK (Java Development Kit)**

The JDK software development kit provided by Oracle Corporation contains tools, libraries, and documentation required for Java development. It includes the following components:

1. **Java Compiler**:  The JDK includes the Java compiler, which translates Java source code (.java files) into bytecode (.class files) that the JVM can execute.
2. **Java Runtime Environment (JRE)**: The JDK includes a JRE, which allows developers to run Java applications during development. It includes the JVM and the core libraries required for running Java programs.
3. **Development Tool:** The JDK provides various development tools like the debugger, profiler, JavaDoc generator, and more, which assist in writing, testing, and debugging Java code.

The JDK is essential for writing and compiling Java programs, while the JRE is sufficient for running them.

**JRE (Java Runtime Environment)**

The JRE is a subset of the JDK required to run Java applications. It includes the JVM and the necessary runtime libraries for executing Java programs. The JRE consists of the following components:

1. **JVM (Java Virtual Machine):** The JVM is the runtime engine that interprets and executes Java bytecode. It provides an abstraction layer between the Java code and the underlying operating system, enabling platform independence.
2. **Java Class Libraries:** The JRE contains the core Java class libraries, which provide pre-compiled classes and methods for common programming tasks. These libraries are essential for running Java applications.

In summary, the JRE is used by end-users who only need to run Java applications without needing development or compilation.

**JVM (Java Virtual Machine)**

The JVM is an integral part of the Java platform and is responsible for executing Java bytecode. It provides a runtime environment in which Java applications can run.

The JVM performs the following key functions:

1. **Bytecode Execution:** The JVM interprets the Java bytecode line by line and executes the corresponding instructions.
2. **Memory Management:** The JVM manages memory allocation and deallocation, including garbage collection, to free up memory occupied by objects no longer in use.
3. **Platform Independence:** The JVM provides platform independence by interpreting the bytecode and translating it into machine code specific to the underlying operating system.

In summary, the JVM is the runtime engine responsible for executing Java applications and ensuring their portability across different platforms.

A screenshot of a computer

Description automatically generatedExecution Flow of a Java Program

1. We write the Java source code in Simple.Java file using an editor or IDE (**integrated development environment**) e.g. *Eclipse* or *IntelliJ Idea*.
2. Program has to be compiled into bytecode. Java compiler (javac) compiles the sourcecode to Simple.class file.
3. This class file can be executed in any platform/OS by JVM (**Java virtual machine**).
4. JVM translates bytecode into native machine code which machines can execute.

**A diagram of a computer system

Description automatically generatedJVM Architecture**

**Class Loader**

1. **Loading**

* **ClassLoader** is a mechanism in Java that loads classes into the **JVM** during runtime.
* The **parent delegation model** ensures classes are loaded in a consistent and secure order.
* Java comes with three primary ClassLoaders: **Bootstrap**, **Platform**, and **Application ClassLoaders**.
* You can create a **Custom ClassLoader** for special use cases, like dynamic or isolated class loading.

ClassLoaders play a crucial role in modular and enterprise Java applications by ensuring the correct classes are loaded at the right time and from the correct source.

**Example:**

Suppose you have a Java program that uses a MyClass class. When MyClass is first referenced in your code, the following happens:

1. **Application ClassLoader** asks the **Platform ClassLoader** to load MyClass.
2. **Platform ClassLoader** asks the **Bootstrap ClassLoader** to load MyClass.
3. The **Bootstrap ClassLoader** tries to load the class. If MyClass is part of the core Java libraries, it loads it; otherwise, it delegates it back to the **Platform ClassLoader**.
4. If the class is not found by the **Platform ClassLoader**, the **Application ClassLoader** loads it from the application's classpath.
5. If class is found by any loader then class is loaded by class loader; else ClassNotFoundException is thrown.
6. **Linking**

After class is loaded by the classloader, linking is performed. A **bytecode verifier** will verify whether the generated bytecode is proper or not. If verification fails we will get a verification error. It also performs the memory allocation to static variables and methods found in the class.

1. **Initialization**

This is the final phase of class loading, here all static variable will be assigned with the original values, and the static blocks will be executed.

JVM Memory Area

The memory area inside JVM is divided into multiple parts to store specific pieces of application data.

* **Method Area** stores class structures like metadata, the constant runtime pool, and the code for methods.
* **Heap** stores all objects that are created during application execution.
* **Stacks** store local variables, and intermediate results. All such variables are local to the thread by which they are created. Each thread has its own JVM stack, created simultaneously as the thread is created. So all such local variable are called **thread-local variables**.
* **PC register** store the physical memory address of the statements which is currently executing. In Java, each thread has its separate PC register.
* Java supports and uses **native code** as well. Java provides a mechanism called **Java Native Interface (JNI)**, which allows Java code to call or be called by native applications and libraries written in other languages like C or C++.

A diagram of a software development

Description automatically generated

JVM Execution Engine

A diagram of a program

Description automatically generated

All code assigned to JVM is executed by an execution engine. The execution engine reads the byte code and executes one by one. It uses both inbuilt *interpreter* and *JIT compiler* to convert the bytecode to machine code and execute it.

Summary:

JVM is the specification for a runtime environment that executes the Java applications. Hotspot JVM is such one implementation of the specification. It loads the class files and uses the interpreter and JIT compiler to convert bytecode into machine code and execute it.

**What is the difference between an interpreter and JIT compiler?**

The interpreter interprets the bytecode line by line and executes it sequentially. It results in poor performance. JIT compiler add optimization to this process by analyzing the code in blocks and then prepare more optimized machine code.

A screenshot of a computer

Description automatically generated

**JAVA MEMORY MODELS**

Java’s memory model plays a important role in the efficient execution of Java applications. A solid understanding of the memory model can help developers avoid performance bottlenecks and memory leaks.

A diagram of a computer system

Description automatically generated

Heap

The heap is a critical component of Java’s memory model, serving as the primary area for storing Java objects. Whenever an object is created using the new keyword in Java, memory for that object is allocated from the heap. This is also where the Garbage Collector operates, reclaiming memory used by objects that are no longer needed, which helps prevent memory leaks and excessive memory usage.

**Structure of the Heap**

The Java heap is subdivided into three main areas:

* **Young Generation:** This is where all new objects are allocated. The Young Generation is further divided into one ‘Eden’ space and two ‘Survivor’ spaces. Objects initially reside in Eden and move to a Survivor space if they remain alive after a garbage collection event.
* **Old Generation:** Objects that have survived several garbage collection cycles in the Young Generation are promoted to the Old Generation. It is designed for objects with a longer lifecycle. The threshold of survival is set by the garbage collector’s policies and can be tuned by the developer.
* **Permanent Generation (only in JVMs before Java 8):** This part of the heap holds metadata such as classes and methods, which do not change frequently.

With Java 8 and later, the introduction of Metaspace has replaced the Permanent Generation.

In the example below, the Customer object, including its name attribute, is allocated to the heap. The heap's ability to dynamically allocate memory makes it ideal for managing complex objects with varying lifespans.

A computer screen shot of a program code

Description automatically generated

**Stack**

Each thread in a Java application has its own stack, which is used for storing short-lived variables and method call information. The stack is smaller in size compared to the heap but is crucial for handling method invocations and storing local variables and intermediate outcomes of expressions.

**How the Stack Works**

When a new method is invoked, a new block called a “stack frame” is created on the stack. This stack frame contains all the local variables, parameters, and the return address of the method. Once the method completes execution, its stack frame is discarded, making this area highly efficient in managing memory that is only needed during a method call.

A screenshot of a computer program

Description automatically generated

This example demonstrates how the stack is used to manage the flow of method execution and the lifecycle of local variables.

Stack Frame and its LifeTime

In Java, when a method is called, a **stack frame** for that method is created in the **stack memory**. This frame contains information such as local variables, parameters, and the return address. Here’s how the lifecycle of a stack frame works:

1. **Method Call (Stack Frame Creation)**:
   * When a method is called, a new stack frame is created at the top of the stack. This frame holds the method’s local variables and parameters, as well as the information needed to return to the calling method after execution.
2. **Method Execution (Active Stack Frame)**:
   * While the method is running, its stack frame remains active at the top of the stack. The JVM uses this frame to manage the method’s execution, including any variables and computations.
3. **Method Completion (Stack Frame Removal)**:
   * Once the method completes (either by returning a value or reaching the end of the method), the stack frame is **popped off the stack**.
   * This removal happens **immediately after the method finishes execution**—the JVM automatically deallocates the memory used by that stack frame. At this point, all the local variables and parameters in that method’s frame are also discarded.
4. **Lifespan of a Stack Frame**:
   * The lifespan of a stack frame is **limited to the duration of the method call**. As soon as the method completes, the stack frame is removed and its memory is deallocated. This means:
     + If the program calls the method multiple times, a new stack frame is created each time, even if the same variables are used within the method.
     + No stack frame or local variable data persists beyond the method’s execution.
5. **End of Program Execution**:
   * Once the program completes (e.g., the main method finishes in a standalone application), all stack frames for any remaining methods are removed as the stack memory is completely cleared.

**Summary:**

* A **stack frame’s lifetime** is only as long as the method’s execution.
* Once the method completes, the stack frame is **immediately removed** from the stack, and its memory is deallocated.
* Local variables within that frame are no longer accessible after the frame is removed, meaning they do not persist beyond the method call.

In short, the stack frame is removed immediately after method execution, freeing up memory for future method calls, and its data does not persist after this point.

Java Metaspace (or Non-Heap Memory or method space)

Metaspace is a non-heap memory area that came into existence with Java 8, replacing the Permanent Generation. It is used to store metadata such as class definitions, method data, and field data. Unlike the heap, Metaspace is allocated out of the native memory, and its size is not fixed but can increase dynamically, which helps prevent the OutOfMemoryErrors that were possible with the Permanent Generation.

**Monitoring and Managing Metaspace**

Since Metaspace can grow dynamically, monitoring its size and usage is crucial to prevent system memory from being exhausted. Java provides several options for monitoring and managing the size of Metaspace, such as -XX:MetaspaceSize and -XX:MaxMetaspaceSize, which allow developers to set initial and maximum Metaspace sizes, respectively.

**Importance of Metaspace**

The introduction of Metaspace enhances performance and scalability by dynamically adjusting memory usage based on application demands, thereby allowing Java applications to manage class metadata more efficiently. This is particularly beneficial in environments where many classes are loaded and unloaded.

Questions (Spring Boot)

1. Hot loading in STS – DevTools Dependency
2. Custom Annotation – in STS
3. What has more priority in STS – configuration file or a Java class with server port?
4. Why constructor injection is preferred than @Autowire in DI?
5. What is the difference bwn Hystrix and Resilience 4J? Which is better?
6. Resilince 4J functionality?
7. RestTemplate functionality?