1. **Java assert Keyword**

The assert keyword in Java is used for **debugging purposes**. It allows you to **test assumptions** in your code during development. If an assert statement fails, it throws an **AssertionError**.It provides an effective way to detect and correct programming errors.

Enable assertion: **java -ea AssertExample**

1. **What happens if main() method is written without String args[]?**

The program will compile, but not run, because the JVM will not recognize the main() method. Remember, JVM always looks for the main() method with a string type array as a parameter.

JVM first executes the static block, if it is present in the program. After that, it searches for the main() method. If the main() method is not found, it gives an error.

1. **Memory Management**

memory management is the process of allocation and de-allocation of objects, called Memory management. Java does memory management automatically.

1. **Heap Area (shared)**

* Stores **objects** and their **instance variables**.
* Managed by the **Garbage Collector (GC)**.
* Divided into:
  + **Young Generation**: New objects, fast collection.
  + **Old Generation (Tenured)**: Long-lived objects.

**2. Stack Area (per thread)**

* Stores **method call frames**, **local variables**, and **function calls**.
* Automatically managed using **LIFO** (last-in, first-out).
* Each thread has its **own stack**.

**3. Method Area / Metaspace (shared)**

* Stores **class-level data** (e.g., class metadata, static variables, method bytecode).
* In Java 8+, PermGen was replaced by **Metaspace**.

1. **Program Counter (PC) Register**

The **Program Counter (PC) Register** is a **small memory space** that holds the **address** of the **next bytecode instruction** to be executed **for each thread**.

**5. Native Method Stack**

* Stores information related to **native (non-Java) methods** (e.g., C/C++).

1. **Java Heap Memory Switches (JVM Options)**

Java provides several **JVM switches** to configure and control **heap memory usage**, which is critical for application performance, garbage collection, and stability.

|  |  |
| --- | --- |
| **Switch** | **Description** |
| **-Xms<size>** | Sets the **initial heap size** |
| **-Xmx<size>** | Sets the **maximum heap size** |
| **-Xmn<size>** | Sets the **size of Young Generation** |
| **-XX:NewSize=<size>** | Initial young generation size (alternative to -Xmn) |
| **-XX:MaxNewSize=<size>** | Max young generation size |
| **-XX:PermSize=<size>** | (Java ≤ 7) Initial Permanent Generation size |
| **-XX:MaxPermSize=<size>** | (Java ≤ 7) Max PermGen size |
| **-XX:MetaspaceSize=<size>** | (Java 8+) Initial **Metaspace** size (class metadata) |
| **-XX:MaxMetaspaceSize=<size>** | (Java 8+) Maximum **Metaspace** size |
| **-XX:SurvivorRatio=<n>** | Sets Eden/Survivor space ratio in Young Gen |
| **-XX:+UseG1GC** | Enables **G1 garbage collector** |
| **-XX:+PrintGCDetails** | Prints detailed GC logs |
| **-XX:+HeapDumpOnOutOfMemoryError** | Dumps memory snapshot on OOM |

1. **Reference Type in Java**

|  |  |  |
| --- | --- | --- |
| **Reference Type** | **Example** | **GC Behavior** |
| **Strong** | Student s = new Student(); | Not eligible for GC unless nullified |
| **Soft** | new SoftReference<>(obj) | Collected only under memory pressure |
| **Weak** | new WeakReference<>(obj) | GC’d eagerly (used in WeakHashMap) |
| **Phantom** | new PhantomReference<>(obj, queue) | Used for advanced memory management (finalization) |

1. **when an object becomes eligible for garbage collection?**

An object in Java becomes **eligible for garbage collection (GC)** when **no live thread can access it anymore**.

— i.e., **there are no references to it** from any reachable part of your code.

1. **Types of Garbage Collector**

|  |  |  |  |
| --- | --- | --- | --- |
| **GC Name** | **Best For** | **Multithreaded** | **Default?** |
| **Serial GC** | Small apps, single-threaded | ❌ No | No |
| **Parallel GC** | High-throughput apps | ✅ Yes | ✅ Java 8 default |
| **CMS (Concurrent Mark-Sweep)** | Low pause time apps | ✅ Yes | ❌ Deprecated |
| **G1 GC (Garbage First)** | Balanced performance & pause time | ✅ Yes | ✅ Java 9+ default |
| **ZGC (Z Garbage Collector)** | Real-time apps, microservices | ✅ Yes | ❌ No |
| **Shenandoah GC** | Ultra-low pause apps | ✅ Yes | ❌ No |

1. **ClassLoader**

The **ClassLoader** is a part of the **Java Runtime Environment (JRE)** responsible for **dynamically loading classes** into the JVM **at runtime**.

|  |  |
| --- | --- |
| **Loader Type** | **Description** |
| **1. Bootstrap ClassLoader** | Loads core Java classes from rt.jar (e.g., java.lang.\*). Written in native code. |
| **2. Extension (Platform) ClassLoader** | Loads classes from lib/ext or java.ext.dirs. |
| **3. System (Application) ClassLoader** | Loads classes from the **classpath** (., jar, or custom -cp path). |
| **4. Custom ClassLoader** | You can create your own class loader by extending ClassLoader. Useful for frameworks or dynamic loading. |

Java ClassLoader is based on three principles: **Delegation**, **Visibility**, and **Uniqueness**.

* **Delegation principle:** ClassLoader follows the **parent delegation model**, where it **forwards the request to the parent ClassLoader** first. The current ClassLoader **loads the** class only if the parent cannot find or load it.
* **Visibility principle:** It allows child class loader to see all the classes loaded by parent ClassLoader. But the parent class loader cannot see classes loaded by the child class loader.
* **Uniqueness principle:** It allows to load a class once. It is achieved by delegation principle. It ensures that child ClassLoader doesn't reload the class, which is already loaded by the parent.

1. **Difference between loadClass() and Class.forName()**

The loadClass() method loads only the class but does not initialize the object. While Class.forName() method initializes the object after loading it. 

1. **Java Decompiler**

A **Java Decompiler** is a tool that converts **compiled .class files (bytecode)** back into **readable Java source code**. This is useful for debugging, reverse engineering, or understanding third-party libraries when source code is unavailable.

1. **Strictfp Keyword**

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable.

1. **Java UUID**

UUID stands for **Universally Unique Identifier**. It's a 128-bit value used to uniquely identify information **without significant risk of duplication**—even across systems & time. Ex. **Transactionid, sessionid**

|  |  |  |
| --- | --- | --- |
| **Version** | **Name** | **Use** |
| 1 | Time-based | Contains timestamp and MAC address (not secure) |
| 3 | Name-based (MD5) | Based on a namespace and name |
| 4 | Random | Random bits, most common |
| 5 | Name-based (SHA-1) | Like v3, but more secure |

1. **JVM**

Key Components of JVM Architecture:

1. **Class Loader Subsystem**
2. **Method Area/Metaspace:** Stores class-level data like: Class metadata, static variable
3. **Heap Area**: Object & instance variable
4. **PC Register (Program Counter):** It is a **small memory space** that holds the **address** of the **next bytecode instruction** to be executed **for each thread**.
5. **Native Method Stack**
6. **Execution Engine**: Executes bytecode instructions.

**JVM Working Process (Simplified)**

1. Java source code is compiled into **bytecode** (.class files).
2. Bytecode is **loaded** into JVM by the ClassLoader.
3. Bytecode is **verified** and prepared.
4. **Execution Engine** runs the bytecode:
   * Interpreter starts executing.
   * Frequently-used code is compiled by JIT into native code.
5. Memory is managed in the **Heap and Stack**.
6. **Garbage Collector** reclaims memory of unused objects.