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) | **Immediately eligible** for GC when no strong/soft references exist. Ex. WeakHashMap |
| **Phantom** | new PhantomReference<>(obj, queue) | Used for advanced memory management (finalization). Used to perform **cleanup after GC**, like **native resource cleanup**. |

**Phantom Reference:**  A **Phantom Reference** is a **special type of reference** that lets you **know when an object is about to be removed from memory (garbage collected)**.

“Hey, the object is really gone. Now it’s safe to clean up whatever it used (like native memory, file handles, etc.).”

Sometimes, your object uses **resources that Java’s garbage collector doesn’t manage**, like:

* **Native memory (off-heap)**
* **Direct byte buffers**
* **File handles or sockets**

You can use PhantomReference to get a **notification when the object is truly dead**, so you can **manually release those resources**.

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.
7. **Sort a List by Name then Age**

=people.stream() .sorted(Comparator.comparingInt(p -> p.age)

.thenComparing(p -> p.name)).collect(Collectors.toList());

The list’s elements must implement **Comparable** or you must supply a **Comparator**.

**Collections.sort(people, Comparator.comparingInt(p -> p.age));**

**Use .thenComparing() for multi-level sorting.**

1. **How to convert byte array to String in Java?**

By using **String class constructor. String str=new String(bytes);**

1. **Java Keystore (JKS)**

The **Java Keystore (JKS)** is a **repository (file)** used by Java to store:

* **Private keys, Public key certificates, Secret keys**

It is commonly used for:

* **SSL/TLS certificates** (HTTPS), **Digital signing, Java application security (e.g., Spring Boot)**

Default File extension: **.jks**

1. **Next() vs NextLine()**

|  |  |  |
| --- | --- | --- |
| **Feature** | **next()** | **nextLine()** |
| **Reads until** | **Whitespace** (space, tab, newline) | **End of line** (\n) |
| **Skips newline?** | ✅ Yes (newline after word is skipped) | ❌ No (reads the whole line including empty) |
| **Returns** | A **single word** | A **whole line** |
| **Common use case** | Reading **words/tokens** | Reading **full lines/sentences** |

1. **CompletableFuture**

A **CompletableFuture** is used for asynchronous programming. It runs a task on a **separate thread than the main thread** and **notifies** the main thread about its progress, completion or failure.

In this way, the **main thread** does not block or wait for the **completion of the task.** Other tasks are executed in parallel.

|  |  |
| --- | --- |
| **Method** | **Purpose** |
| **runAsync()** | Runs Runnable (no return) in background |
| **supplyAsync()** | Runs Supplier<T> in background and returns result |
| **thenApply()** | Transform result(means customization on result) |
| **thenAccept()** | Consume the result without returning |
| **thenRun()** | Run something after result is ready |
| **thenCombine()** | Combine two futures |
| **handle()** | Handle result or exception |
| **exceptionally()** | Handle only exceptions |

CompletableFuture.**supplyAsync**(() -> "hello").**thenApply**(s -> s.toUpperCase());

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Takes Input?** | **Returns Value?** | **Purpose** |
| **thenApply()** | ✅ Yes | ✅ Yes | Transform and return value |
| **thenAccept()** | ✅ Yes | ❌ No | Consume value, no return |
| **thenRun()** | ❌ No | ❌ No | Run task after completion |

**Future** : A CompletableFuture is an extension to Java's Future API which was introduced in Java 8.

**Limitations of the Future**

* No built-in support for **chaining. Future don’t have thenApply() & ThenSupply() etc.**
* No way to **manipulate result asynchronously** . ex. thenApply()
* Future has not any exception handling.
* We cannot combine multiple futures.

1. List<Integer> list = Arrays.asList(5,9,14);
2. list.stream().map(num->CompletableFuture.**supplyAsync**(()->getNumber(num))).map(CompletableFuture->CompletableFuture.**thenApply**(n-

>n\*n)).map(t->t.**join**()).forEach(s->System.out.println(s));  }

|  |  |
| --- | --- |
| **Method** | **Description** |
| **get()** | Waits for the result (blocks if needed) |
| **get(timeout, unit)** | Waits for a limited time |
| **isDone()** | Checks if task is completed |
| **cancel(true)** | Attempts to cancel the task |
| **isCancelled()** | Checks if task was cancelled |

|  |  |  |
| --- | --- | --- |
| **Feature** | **Future** | **CompletableFuture** |
| **Blocking get()** | ✅ Yes | ✅ Yes |
| **Non-blocking chaining** | ❌ No | ✅ Yes (thenApply) |
| **Combine multiple futures** | ❌ No | ✅ Yes |
| **Exception handling** | Manual try-catch | Built-in methods |
| **Better for complex workflows** | ❌ No | ✅ Yes |

1. **How to Sort HashMap by Value**

= map.entrySet() .stream().sorted(Map.Entry.comparingByValue())

.collect(Collectors.toMap(Map.Entry::getKey,Map.Entry::getValue,

(e1, e2) -> e1, // merge function

LinkedHashMap::new // maintains insertion order));

|  |  |  |
| --- | --- | --- |
| **Feature** | **map()** | **collect()** |
| Purpose | **Transform elements** in the stream | **Gather results** into a collection or summary |
| Returns | A **new Stream** | A **final result** (List, Map, Integer, etc.) |
| Input | A function (Function<T, R>) | A collector (Collector) like toList() |
| Chainable | ✅ Yes | ❌ No (ends the stream) |

1. **Differences Between Array and ArrayList**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Array** | **ArrayList** |
| **Size** | **Fixed** (set at creation) | **Dynamic** (grows/shrinks automatically) |
| **Type** | Can store **primitives** and objects | Can store **only objects** |
| **Resizing** | ❌ Manual (create new array) | ✅ Automatic |
| **Add/Remove elements** | ❌ Not supported directly | ✅ add(), remove() methods available |
| **Methods available** | Only basic features (length, index) | Rich API (add(), contains(), etc.) |
| **Performance** | Faster (less overhead) | Slightly slower (more overhead) |
| **Loop-friendly** | ✅ Works with for/foreach loop | ✅ Works with loops and streams |
| **Type Safety (Generic support)** | ❌ No generics | ✅ Generic type-safe |
| **📄 toString()** | Shows object address | Shows contents |

1. **How to Compare Two ArrayLists**

First Sort using **Collection**.**sort**() then compare using **equal**()

1. **How to Remove Special Characters from String**

you can use **regular expressions (RegEx)** with String.replaceAll().

1. **ConcurrentModificationException**

**ConcurrentModificationException** is a runtime exception that occurs when a collection is modified while being iterated.

|  |  |
| --- | --- |
| **Situation** | **Safe?** |
| **Modifying collection in for-each** | ❌ No |
| **Using iterator.remove()** | ✅ Yes |
| **Using removeIf()** | ✅ Yes |
| **Using CopyOnWriteArrayList** | ✅ Yes (for concurrency) |
| **Using List.remove() during stream** | ❌ No |

1. **How to Print ASCII Value in Java**

To **print ASCII values** of characters in Java, you can simply **cast the character to an int**

1. **Program to Swap Two Numbers**

x = x\*y;  x= x+y  a = a + b;

y = x/y;  y= x-y b = a.substring(0, a.length() - b.length());

x = x/y;  X=x-2 a = a.substring(b.length());

1. **Program to Find GCD of Two Numbers**
2. for(**int** i = 1; i <= x && i <= y; i++)  {
3. **if**(x%i==0 && y%i==0)  { gcd = i; }}

27. **Shift Operators in Java (Operator Shifting)**

* **Left Shift (<<)** int b = a << 1; // output 10 Formula a \* 2ⁿ
* **Right Shift (>>)** int b = a >> 1; // output 10 Formula a / 2ⁿ floor value

**28. Perfect Number Program**

A number whose **sum of factors** (excluding the number itself) is equal to the number is called a **perfect** **number**.

Ex. 28= 1+2+4+7+14

1. **PipedInputStream and PipedOutputStream**

PipedInputStream and PipedOutputStream are **Java I/O classes** that allow two threads to **communicate with each other** via a **pipe** (like a data tunnel).

They are used for **thread-to-thread communication** where:

* One thread **writes data** to a PipedOutputStream
* Another thread **reads data** from the connected PipedInputStream

**pout.connect(pin);** *//connecting the streams*

1. **How to avoid deadlock**

* Avoid Unnecessary Locks:
* Avoid Nested Locks:
* Using Thread.join() Method:
* Use Lock Ordering:
* Lock Time-out:

1. **SortedSet**

SortedSet is an interface in Java that extends Set and ensures that elements are **sorted in natural order. Treeset implemented it.**

1. **Framework vs. Library in Java**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Library** | **Framework** |
| **Definition** | A collection of reusable functions/classes | A complete structure to build and organize applications |
| **Control Flow** | **You call** the library functions | **Framework calls** your code (Inversion of Control) |
| **Flexibility** | More flexible — use only what you need | Less flexible — follows specific structure/rules |
| **Learning Curve** | Generally easier to start with | May have a steeper learning curve |
| **Ex. in Java** | Apache Commons, Jackson, JDBC, Guava | Spring Framework, Hibernate, Struts, JSF |
| **Main Goal** | Helps with specific tasks (e.g., parsing, logging) | Provides a foundation for entire application flow |

1. **Why Java is Secure**

|  |  |
| --- | --- |
| **Feature** | **Security Benefit** |
| **No pointers** | No direct memory access (safer) |
| **Bytecode verifier** | Before running, the JVM’s **class loader and bytecode verifier** check the bytecode for: **Access violations**, **Type mismatches**, etc |
| **Security Manager & Policy** | Fine-grained runtime permissions |
| **Sandboxing** | Run untrusted code in a restricted environment (sandbox). |
| **Class Loader Isolation** | Prevents class spoofing |
| **Automatic Garbage Collection** | Avoids memory corruption vulnerabilities |
| **Strong Typing** | Catches bugs at compile/runtime |
| **java.security , Cryptographic APIs** | Supports secure communication ex. Digital signatures |

1. **What is Java Bytecode?**

* Java bytecode is the set of instructions for the Java Virtual Machine.
* Java bytecode is not **JDK/JVM** dependent, but **JRE** dependent.

1. **BiConsumer Interface**

It is a **functional interface** that **accepts two input arguments** and **returns no result**.

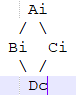
1. BiConsumer<String, String> biCon = BiConsumerInterfaceExample::**ShowDetails**;
2. biCon.**accept**(“Emp”, "Student");   // Where showdetails is a method i.e. accept 2 args

BiConsumer<String, Integer> combined = biCon .**andThen**(2ndBiconsumer);

combined.**accept**("X", “Y”); // Combine Two BiConsumers Using andThen()

1. **What is the Diamond Problem in Java?**

The **Diamond Problem** is a **multiple inheritance ambiguity** that arises when **a class inherits from two classes (or interfaces) that both inherit from a common superclass/interface**, leading to confusion about **which method to inherit**.



Java **doesn’t support multiple inheritance of classes**, so it **avoids** the diamond problem **by design**.

**With Interfaces: Yes, but Resolved**

Java **allows multiple inheritance of interfaces**, and the diamond problem **can occur**, but Java handles it **deterministically** using default method resolution

**How Java Resolves It (Rules)**

1. **Class wins over interface**  
   If a method exists in both a superclass and an interface, the **class method is chosen**.
2. **Subinterface wins**  
   If both interfaces provide the same default method, the **most specific subinterface** method is used.
3. **Conflict? You must override it**  
   If multiple interfaces provide conflicting default methods → **compiler forces you to override it**.
4. **DoubleBuffer**

DoubleBuffer is a buffer class in Java that handles sequences of primitive double values. It’s part of the Java NIO package and is useful when working with **large, high-performance data transfers**, like in I/O or memory-mapped files.

|  |  |
| --- | --- |
| **Method** | **Description** |
| **allocate(int capacity)** | Allocates a new DoubleBuffer with the given size |
| **wrap(double[] array)** | Wraps an existing double[] array in a buffer |
| **put(double d)** | Writes a double at the current position and specific |
| **get()** | Reads a double from the current position and a specific |
| **flip()** | Switches the buffer from **write mode to read mode** |
| **clear()** | Clears the buffer for writing again |
| **rewind()** | Re-read from the beginning |
| **compact()** | Copies unread data to the beginning and prepares for writing |
| **remaining()** | Returns remaining elements to be read. Same as hasNext() of iterator |
| **mark()** | Marks the current position |
| **reset()** | Resets to the last marked position |
| **duplicate()** | Creates a new buffer that shares content but has its own position, limit, mark etc. |

**arrayOffset**(): Returns the **offset (index)** within the backing array where the buffer’s data **starts**.

int[] numbers = {10, 20, 30, 40, 50};

IntBuffer buffer = IntBuffer.wrap(numbers, 2, 3); **// Starts at index 2, length 3**

**buffer.arrayOffset(); // Output: 2**

**slice**() : The slice() method is used to create a **view buffer** as the original buffer, but **only for the remaining portion**, starting from the current position to the limit.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Shared content | ✅ Yes, shares the same data with the original buffer |
| Independent position | ✅ Yes, the new buffer starts at position 0 |
| Writable | ✅ Yes (unless original is read-only) The original is in write mode, the slice cannot write |
| Affects original | Changes to the data are **visible in both buffers** |

1. **Could Not Find or Load Main Class Error**

|  |  |
| --- | --- |
| ❌ Cause | **✅ Solution** |
| **Class name typo** | Make sure the class name is typed exactly with the correct casing. |
| **No main(String[] args) method** | Ensure you have: public static void main(String[] args) |
| **Wrong directory (not in classpath)** | Run Java command from correct folder or use -cp (classpath). |
| **Missing .class file** | Compile it first with javac YourClass.java |
| **Package declaration but wrong path** | Use the right folder or package-qualified class name |
| **File compiled with different class name** | Ensure file and class name match (Hello.java → class Hello) |
| **Using wrong java or javac version** | Check with java -version and javac -version |
| **Dependencies missing from the CLASSPATH.** | check dependency |

1. **Compare Two Arrays**

Arrays.sort(), Arrays.equals(), Arrays.deepEquals()

**Arrays.deepEquals():** It is used to **compare two arrays deeply**, meaning it can check equality for **nested arrays (multi-dimensional arrays)**, **arrays of objects**, and **arrays of arrays**.

1. **CyclicBarrier in Java Multithreading**

CyclicBarrier is a **synchronization aid** used in **multithreaded programming** to make a group of threads **wait for each other** to reach a **common barrier point**.

once all threads reach the barrier, it runs a task and resets automatically for reuse.

1. **Structure of Java Program**

* Documentation Section
* Package Declaration
* Import Statements
* Interface Section
* Class Definition
* Class Variables and Variables
* Main Method Class
* Methods and Behaviors

1. **What is a Java Interpreter?**

A **Java interpreter** is a **runtime component** that **reads .class files (bytecode)** and executes the instructions line by line.

|  |  |
| --- | --- |
| **Interpreter** | **Compiler** |
| It translates the code instruction by instruction. | It translates the entire program at once. |
| Its execution is slower. | Its execution is faster. |
| Its compile time is less. | It takes more time to compile the code. |
| It does not generate the intermediate object code. | It generates the intermediate object code. |
| It compiles the program until an error is found. | All errors show once at the end of compilation. |

1. **Program to Determine Whether a Given String of Parentheses (Single Type) is Properly Nested**

* "(())" is properly nested.
* "(()))" is not properly nested.

Write a program using a stack. With push & pop method

1. **Program to Find the Frequency of All Duplicate Elements in an Array**

 hashMap.put(num, hashMap.**getOrDefault**(num, 0) + 1);

1. **Program to Find the Minimal Average of Any Slice Containing at Least Two Elements**

double minAvg = Double.MAX\_VALUE;

double avg2 = (A[i] + A[i + 1]) / 2.0; // Slice of 2

double avg3 = (A[i] + A[i + 1] + A[i + 2]) / 3.0; // Slice of 3

1. **What is a token in Java?**

The [Java compiler](https://www.tpointtech.com/compiler/java) breaks the line of code into text (words) are called **Java tokens**. These are the smallest elements of the [Java program](https://www.tpointtech.com/java-programs).

**Ex**. **Keywords, Identifiers, Literals, Operators, Separators, Comments, etc**

1. **StringTokenizer**

The StringTokenizer class is used to **break a string into tokens**, usually based on **delimiters** like spaces, commas, or other characters.

1. **Tail Recursion**

**Tail Recursion** is a special kind of recursion where the **recursive call is the last operation** in the function.  
This allows some compilers (like in Scala or C) to optimize it and **reuse the same stack frame**, preventing stack overflow.

**Why Tail Recursion Matters?**

* **More memory efficient** (in languages that optimize it)
* **Avoids stack overflow**
* Java **does not optimize** tail recursion natively — but it’s still good practice when designing recursive logic

1. **Busy Waiting in Multithreading (Java)**

**Busy Waiting** is a multithreading technique where a thread **repeatedly checks a condition** in a loop **without releasing the CPU**, instead of waiting passively.

**Better Alternative**: Use wait() / notify()

1. **What Is JIT?**

The JVM compiles **frequently-used bytecode methods** into **native machine code** while the program is running.

**How JIT Works**

* **You compile Java code:** javac Hello.java → Hello.class (bytecode)
* **You run the program**: java Hello
* JVM starts interpreting bytecode.
* If a method is called **frequently** (called a "hotspot"): JIT compiler compiles it into **native machine code**. That method then runs faster from that point on.

1. **Final vs. Immutability**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **final** | **Immutability** |
| **Definition** | Ensures the reference or variable cannot be reassigned. | Ensures the object’s internal state cannot be changed after creation. |
| **Application** | Applied to classes, methods, and variables. | Applies to objects to maintain an unchangeable state. |
| **Usage for Variables** | Prevents reassignment after initialization. | Ensures the values inside the object don’t change. |
| **Usage for Methods** | Prevents method from being overridden in subclasses. | Not directly related to methods; focuses on object data. |
| **Usage for Classes** | Final class cannot be extended. | Immutability can use final but is not solely defined by it. |
| **Impact on Object State** | Reference is fixed; object’s internal state may still be changed. | Both reference and internal state are fixed and unmodifiable. |
| **Reference vs. State** | Reference is constant; internal state can still change. | Both reference and state are fixed; object is entirely immutable. |

1. **Reverse a String Using Recursion**

return reverseString(str.substring(1))+str.charAt(0);

1. **Generics Vs. Wildcard**

**Generics** are primarily used to provide type safety by enabling the compiler to impose **type checks during compilation**. By doing this, casting is not necessary, and the chance of a runtime [ClassCastException](https://www.tpointtech.com/classcast-exception-in-java) is decreased.

**Wildcards:** Wildcards are special parameters used with generics **to represent an unknown type**. Wildcards are represented by a question mark (?).

|  |  |  |
| --- | --- | --- |
| **Feature** | **Generics** | **Wildcards** |
| **Definition** | Specifies a concrete type (e.g., List<String>) | Represents an unknown type (e.g., List<?>) |
| **Purpose** | Defines a specific type for a class or method | Provides flexibility when the exact type is not needed |
| **Type Safety** | Enforces strict type safety at compile-time | Offers less strict safety but more flexibility |
| **Usage** | Used in class, interface, & method definitions | Typically used in method parameters or return types |
| **Syntax** | Defined using angle **brackets <>** | Represented by a question **mark ?** |
| **Bounds** | Can use **<T extends Number> or <T super Number>** | **Uses <? extends T> and <? super T>** |
| **Reading & Writing** | Can **read and write** using the exact type | <? extends T> → can read, <? super T> → can write |
| **When to Use** | When you know the exact type. | When you don’t need to know the exact type |
| **Code Example** | List<String> list = new ArrayList<>(); | void printList(List<?> list) |

1. **Program to check whether an array is a permutation**

An array of **length n** is a permutation if: It contains all integers from 1 to n

* Each number appears exactly once

**Sol: using set, in for loop:** if (!set. add(num)) {return false;}

1. **Stack vs Heap in Java**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Stack Memory** | **Heap Memory** |
| **Storage** | Stores primitive values, method calls, & references | Stores all Java objects (new) |
| **Access** | LIFO (Last In, First Out) — fast & structured | Random access — slightly slower |
| **Scope** | Local to a thread | Shared across all threads |
| **Lifetime** | Short-lived (ends when the method ends) | Long-lived (until GC collects) |
| **Management** | Automatically managed (stack frame pushed/popped) | Managed by Garbage Collector |
| **Thread Safety** | Thread-safe (each thread has its own stack) | Not thread-safe by default |
| **Performance** | Faster memory allocation & deallocation | Slower due to GC and shared access |
| **Example** | int x = 10; (primitive), method() call | new Employee() |
| **Error Ex.** | StackOverflowError if stack exceeds limit | OutOfMemoryError if heap is full |