**1. Core Java Concepts**

These questions test your understanding of the fundamental concepts of Java.

* [**OOP Principles**:](#oops)
  + Can you explain the four pillars of Object-Oriented Programming (OOP)?
  + What is the difference between abstract class and interface in Java?
  + What is polymorphism? Can you give an example?
  + How does inheritance work in Java? Can you explain method overriding and method overloading?
* [**Data Types**](#Datatypes):
  + What is the difference between == and equals() in Java?
  + What is the difference between String, StringBuilder, and StringBuffer?
* **Exception Handling**:
  + What is the difference between checked and unchecked exceptions?
  + How do you create your own custom exception?
  + What are try-with-resources and finally used for? Can you give an example?
* **Garbage Collection**:
  + Can you explain how garbage collection works in Java?
  + What is the difference between final, finally, and finalize()?
* **Concurrency and Multithreading**:
  + How does synchronization work in Java?
  + What is the difference between synchronized methods and synchronized blocks?
  + What is the volatile keyword? How does it work in multi-threaded environments?
  + What are Thread, Runnable, and Callable interfaces? Explain how to use them.

**2. Advanced Java Concepts**

* **Collections Framework**:
  + What is the difference between ArrayList and LinkedList?
  + How does a HashMap work internally?
  + Explain the concept of **hashing** and how **collision resolution** works in a HashMap.
  + What is the difference between HashMap and ConcurrentHashMap?

How do they handle concurrent access?

* + Can you explain the difference between HashMap and TreeMap? In which scenario would you use one over the other?
  + Explain the concept of LinkedHashMap. What makes it different from HashMap?

When would you use a LinkedHashMap?

* + What is the HashSet? How does it ensure uniqueness?
  + How does the PriorityQueue work?
  + What is the underlying data structure of a PriorityQueue?
  + What is the Comparable interface and Comparator interface? Can you explain their differences?
* **Java Streams and Lambda Expressions**:
  + Can you explain the purpose of lambda expressions in Java and advantages? Can you provide an example?
  + What is the difference between a Stream and a Collection in Java?
  + When should you use Stream over Collection?
  + What is a Stream in Java, and how does it work? What are some operations you can perform on a Stream?
  + What is the difference between map() and flatMap() in Java Streams?
  + What is the purpose of the default methods in Java interfaces (introduced in Java 8)?

How do default methods improve backward compatibility?

* + What is a Function, Consumer, and Supplier in Java 8?

Can you explain how they work and when they are used?

* + Explain Optional in Java.
  + How does it help avoid NullPointerExceptions, and when would you use it?
* **JVM Internals**:
  + What are the different memory areas in the JVM (Heap, Stack, Method Area, etc.)?
  + What is the role of the Java ClassLoader? Can you explain the different types of class loaders?
  + How does Java handle memory management, and what is the role of the Garbage Collector?
  + What is the role of the Garbage Collector in Java?
  + How does the GC work, and what are the different types of Garbage Collection algorithms in Java?
  + What is the difference between JVM and JRE? Can you explain the roles of JVM, JRE, and JDK?
  + What are annotations in Java?Can you explain common annotations like @Override, @SuppressWarnings, @Transactional, etc.?

**3. Design Patterns**

An interviewer may ask about the **design patterns** you have worked with.

* **Common Patterns**:
  + What are some design patterns you have used in your Java projects? Can you explain a scenario where you used one?
  + Can you explain the Singleton pattern with an example?
  + What is the Factory pattern, and when would you use it?
  + What is the Observer pattern? How is it used in event-driven programming?

**4. Java Frameworks and Libraries**

If you have worked with any frameworks or libraries (Spring, Hibernate, etc.), you may be asked questions related to them.

* **Spring Framework**:
  + What is the Spring Framework, and how does Dependency Injection work in Spring?
  + What is the difference between BeanFactory and ApplicationContext?

When would you use each in a Spring application?

* + What is the difference between @Autowired and @Inject annotations?
  + What are the different types of Spring AOP (Aspect-Oriented Programming)?

Explain how cross-cutting concerns are handled in Spring.

* + What is Spring Boot, and how does it simplify application development?
* **Hibernate/JPA**:
  + What is JDBC in Java? How does JDBC work, and what are the steps to connect to a database using JDBC?
  + What are the differences between JDBC and JPA?How does JPA improve database access compared to using raw JDBC?
  + Explain Hibernate and how it works with Java.How does Hibernate implement Object-Relational Mapping (ORM)?
  + What is the Lazy Loading and Eager Loading in Hibernate?How do they affect performance in Hibernate-based applications?
  + What is EntityManager in JPA?How do you perform CRUD operations using EntityManager in a JPA-based application?
  + What are Transactions in Hibernate and JPA?How do transactions work in Java-based persistence layers?
  + How does Hibernate handle ORM (Object-Relational Mapping)? What are Entity, Session, and Transaction in Hibernate?

**5. Problem-Solving and Coding Challenges**

Interviewers often give coding problems to test your ability to solve problems efficiently.

* **Example Coding Problems**:
  + Write a function to reverse a string in Java.
  + Given an array of integers, find two numbers that add up to a specific target.
  + Implement a simple version of a linked list or a binary tree.
  + Find the missing number in an array containing numbers from 1 to n (inclusive) with one number missing.

**6. System Design**

For candidates with 3 years of experience, they might ask some **high-level system design** questions to understand your problem-solving and architectural skills.

* **Example Questions**:
  + Design a URL shortening service like Bitly.
  + How would you design a rate-limiting service (e.g., for an API)?
  + How would you design a system for managing online orders?

**7. Version Control**

If you use version control like **Git**, you may be asked about your experience with it.

* **Git Questions**:
  + What is the difference between git pull and git fetch?
  + How do you resolve merge conflicts in Git?
  + How do you create a branch and merge it back into the main branch in Git?

**8. Testing**

Since Java developers are expected to be familiar with writing unit tests, especially in frameworks like **JUnit**, you may be asked about testing.

* **Testing Questions**:
  + How do you write unit tests in Java?
  + What is the difference between **JUnit 4** and **JUnit 5**?
  + What is mocking, and how do you mock dependencies in unit tests (e.g., using **Mockito**)?

**9. Maven**

**Tips for Preparation:**

1. **Review Key Concepts**: Brush up on core Java concepts, OOP principles, and advanced topics.
2. **Practice Coding**: Solve coding problems regularly (use platforms like LeetCode, HackerRank, or CodeSignal).
3. **Understand Frameworks**: If you’ve worked with frameworks like Spring or Hibernate, be sure you understand how they work.
4. **Explain Your Work**: Be prepared to explain your previous projects and the challenges you faced.
5. **Think Aloud**: During coding interviews, explain your thought process clearly as you solve the problem.

The four pillars of Object-Oriented Programming (OOP) are:

**Encapsulation, Abstraction, Inheritance, Polymorphism**

1. **Encapsulation**:
   * Encapsulation is the concept of bundling the data (attributes) and the methods (functions) that operate on the data into a single unit, called a class.
   * It refers to the concept of **hiding the internal state** of an object and **restricting access** to it, only allowing modifications through well-defined methods. This is done by making the fields of a class **private** and providing **public getter and setter methods** to access and update those fields.
   * It also involves restricting direct access to some of an object's components, which is typically done through access modifiers like private, protected, and public. This ensures that an object's internal state is hidden from the outside world, and it can only be modified through well-defined methods (also called getters and setters) which allow controlled access to the private attributes of an object.

In Java, encapsulation is typically achieved by:

1. **Declaring the class variables as private** to restrict direct access from outside the class.
2. Providing **public getter and setter methods** to allow controlled access and modification of these variables.
   * **Example**: A Car class might have private attributes like engineStatus and fuelLevel, and public methods like startEngine() and checkFuelLevel() to interact with them.
3. **Abstraction**:

* Abstraction refers to hiding the complex implementation details and showing only the essential features of an object. The goal is to provide a simplified interface that makes it easier to interact with the object.
* It allows developers to work with high-level concepts while ignoring low-level details.
* **Example**: A RemoteControl class can have a method pressButton(), but users don't need to know how it sends a signal to the TV, they just interact with the high-level functionality of pressing buttons

Abstraction is typically achieved using two types of Abstraction:

1. **Abstract Classes**
2. **Interfaces**

**1. Abstract Classes**

An **abstract class** in Java is a class that cannot be instantiated on its own. It is meant to be subclassed by other classes. An abstract class can have both **abstract methods** (methods without a body) and **concrete methods** (methods with a body).

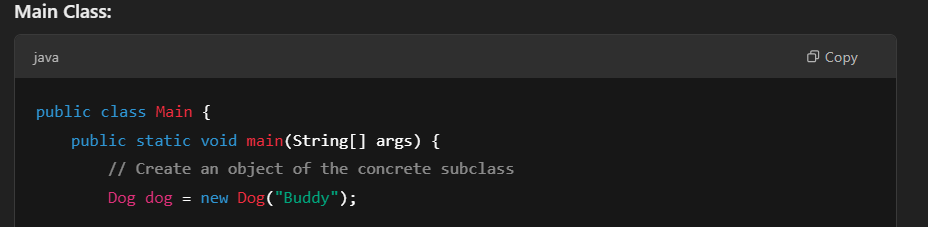
* **Abstract methods** define the behavior of subclasses without specifying the exact implementation.
* **Concrete methods** can provide common functionality that subclasses can inherit.
* **Abstract class** can have **constructors.**
* Abstract classes **cannot be instantiated** directly.
* Even though we can't create an object of the abstract class directly, the **constructor of the abstract class** will be invoked when an object of a **concrete subclass** is created. It helps initialize the common properties of the abstract class. If the abstract class has a constructor, **it will be called from the subclass constructor**, either explicitly or implicitly.
* The abstract class's **fields/properties** are initialized via the abstract class's constructor, but this initialization happens **through the subclass** constructor by invoking the abstract class constructor using **super().**
* **A top-level abstract class** can only be **public or default**
* **Inner abstract classes** can be declaredwith any access modifier**, including private or protected.**
* Can have **private, protected, default or public** methods.

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

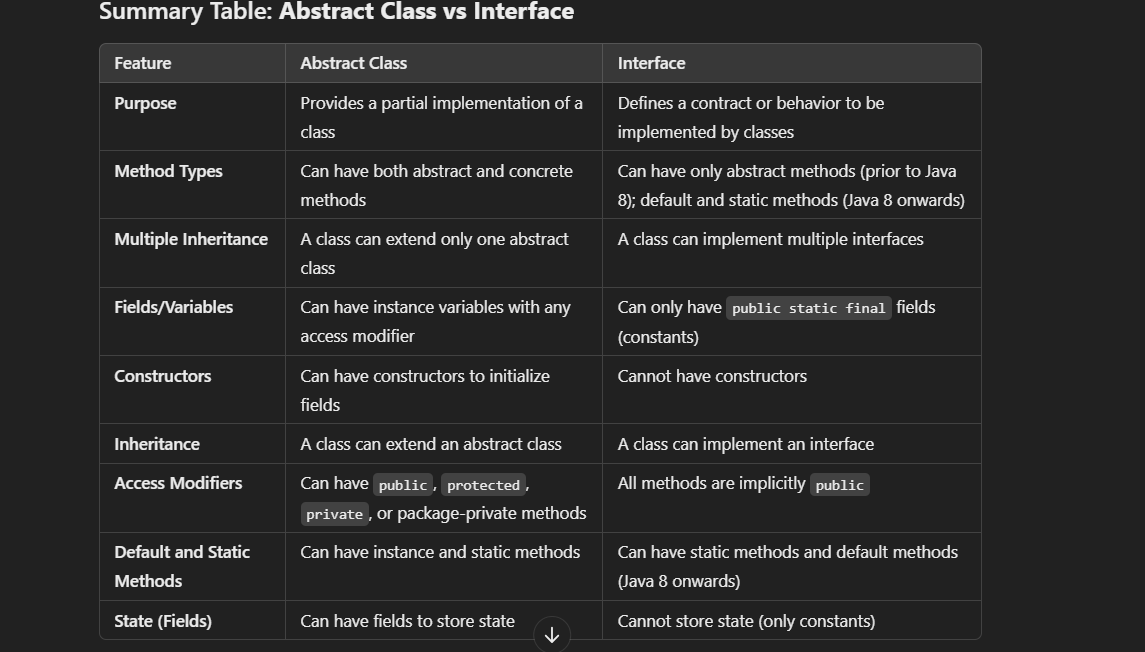


A screen shot of a computer program

AI-generated content may be incorrect.

**Interface**:

* Prior to Java 8, an interface can only have **abstract methods** (without implementation).
* Since Java 8, an interface can have **default methods** (methods with a body) and **static methods**.
* All methods in an interface are **implicitly public**, even if you dont specify public.
* All fields in an interface are **implicitly public, static, and final**. This means they are constants, and they must be initialized when declared.
* Interface **cannot have instance variables** because interfaces are meant to define behavior, not hold state.
* An interface cannot have **constructors** because you cannot instantiate an interface directly. An interface only defines methods/behaviour, not state.



1. **Inheritance**:

* Inheritance is a mechanism where one class (called a subclass or derived class) can inherit attributes and methods from another class (called a superclass or base class). This allows for code reuse and the creation of more specialized classes.
* A subclass can add its own specific attributes and methods or override the ones it inherited from the superclass.
* **Example**: A Dog class might inherit from a more general Animal class, meaning it gets all the characteristics and behaviors of Animal (like eat() and sleep()), but it can also have its own methods like bark().

1. **Polymorphism**:

The ability of an object to take on many forms.

There are two types of polymorphism in Java:

1. **Compile-time Polymorphism (Static Polymorphism)** — Achieved through method **overloading** and **operator overloading**.
2. **Runtime Polymorphism (Dynamic Polymorphism)** — Achieved through method **overriding**.

**Method Overloading** occurs when two or more methods in the same class have the **same name** but differ in the **number of parameters** or the **type of parameters**. This is resolved at **compile-time**, hence it is also known as **Static Polymorphism**.

The correct method to be called is determined at **compile-time** based on the number and types of arguments passed.

A screenshot of a computer program

AI-generated content may be incorrect.

**Method Overriding** is when a subclass provides a specific implementation of a method that is already provided by its superclass. The method in the superclass is **redefined** (overridden) in the subclass to give a different behavior. This is a form of **Dynamic Polymorphism**, because the method to be invoked is determined **at runtime**, depending on the actual object type.

 A **subclass provides its own version** of a method defined in a parent class.

 Happens using **inheritance**.

 Java determines the method to call **at runtime** using the object type (not the reference type).

A screenshot of a computer program

AI-generated content may be incorrect.

**Difference Between == and equals() in Java**

In Java, both == and equals() are used for comparison, but they are used in different contexts and behave differently. Here's a detailed explanation of both:

**1. == Operator**

* The == operator is used for **reference comparison** (for objects) or **value comparison** (for primitive types).
* When comparing **primitive types** (e.g., int, char, boolean), == checks if the **values** are the same.
* When comparing **objects**, == checks if both references point to the **same memory location** (i.e., it checks if both references point to the exact same object in memory).

**Example 1: Comparing Primitives with ==:**

int a = 10;

int b = 10;

if (a == b) {

System.out.println("a and b are equal"); // Output: a and b are equal

}

* Here, == compares the actual **values** of a and b, and since they are both 10 and these are primitives, the condition evaluates to true.

**Example 2: Comparing Objects with ==:**

String str1 = new String("Hello");

String str2 = new String("Hello");

if (str1 == str2) {

System.out.println("str1 and str2 are the same object");

} else {

System.out.println("str1 and str2 are different objects");

}

**Output**:

str1 and str2 are different objects

* Although str1 and str2 have the same value, == checks if they point to the **same object** in memory. In this case, str1 and str2 are two different objects created using the new keyword, so == returns false.

**2. equals() Method**

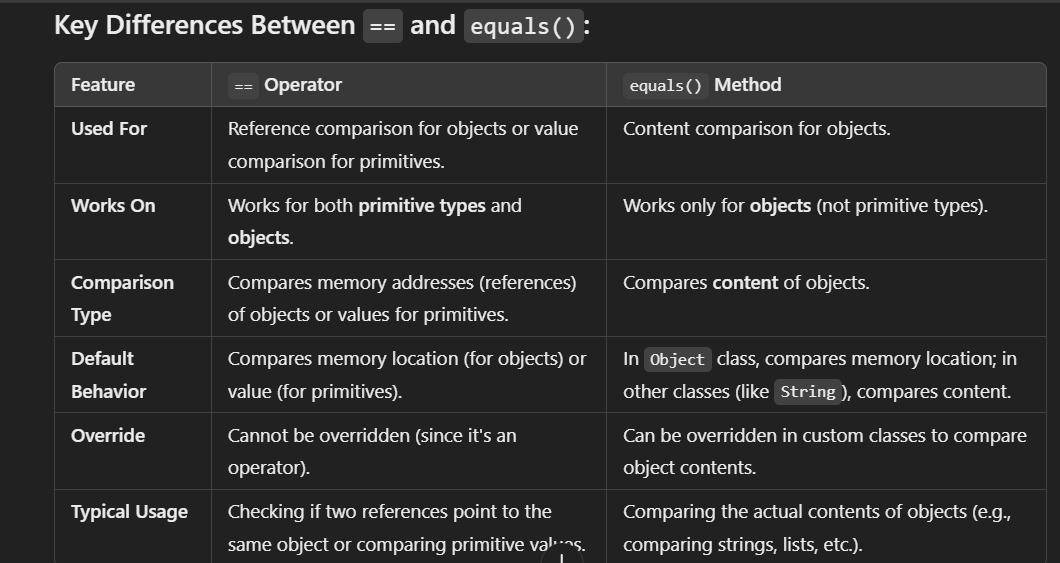
* The equals() method is intended for **content comparison**. It is a method defined in the Object class and is **overridden** in many classes (such as String, Integer, Date, etc.) to compare the actual contents of the objects.
* The default implementation of equals() in the Object class behaves like ==, comparing object references.
* Many classes (like String, Integer, List, etc.) override equals() to compare the **contents** of the objects.

**Example 1: Comparing Strings with equals():**

A screenshot of a computer

AI-generated content may be incorrect.

In this case, equals() compares the **contents** of the strings (i.e., the sequence of characters inside), and since both strings contain "Hello", the method returns true.



**String :::**

**String** is a **class**, but it is also an **object** when you instantiate it. It is immutable. This means that once a String object is created, its value cannot be changed. If you try to modify a String, a new String object is created instead.

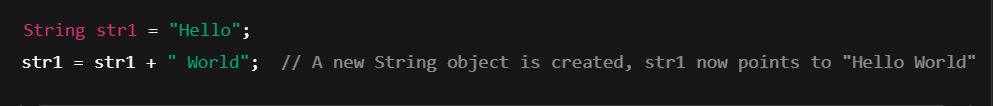
String objects are created either directly through string literals (which are stored in the **String Pool**) or via the new keyword (which creates a new String object on the **heap**).

Since strings are immutable, every time you perform a modification (such as appending or replacing characters), a new String object is created, which can be inefficient when performing many string manipulations in a loop or large-scale operations.

Since it’s immutable, it is inherently thread-safe.

Slower when performing a lot of modifications (like appending in a loop).

Best for constant or rarely changing strings.

  
**So what happens here?**

* When you write s = "Hello";, a String object is created with the value "Hello" in the String pool(in heap) and s points to the string literal "Hello".
* Then, s + " World" is not modifying the existing "Hello" object.
* Instead, it creates a new String object with the value "Hello World" in the heap and assigns it back to the variable s.

**🧠** The original "Hello" object still exists in memory (if not garbage collected), and it's unchanged. That's what immutability means.

** ✅** The reference s is updated to point to a new string ("Hello World").

 ❌ But the original string "Hello" is not changed in memory, just not referenced by any variable anymore.

**StringBuilder:** StringBuilder is mutable and is used for string manipulation. It allows changes to the string without creating a new object for each modification, making it more efficient for repeated string modifications.

* **Mutable**: Can be modified without creating new objects.
* **Not Thread Safe**: Not synchronized, so not safe for multi-threaded environments.
* **Performance**: Faster than String Buffer and String when used in a single-threaded context.
* **Usage**: Ideal for building strings in loops or when performance matters and thread safety isn’t required.

**Eg**:

****

**Also this is a common method to initialize a StringBuilder with an existing string.java**

String initialString = "Hello";

StringBuilder sb = new StringBuilder(initialString); // Initializes with an existing string

sb.append(" World!");

System.out.println(sb); // Output: Hello World!

Use **toString()** for conversion of mutable to immutable type String in java

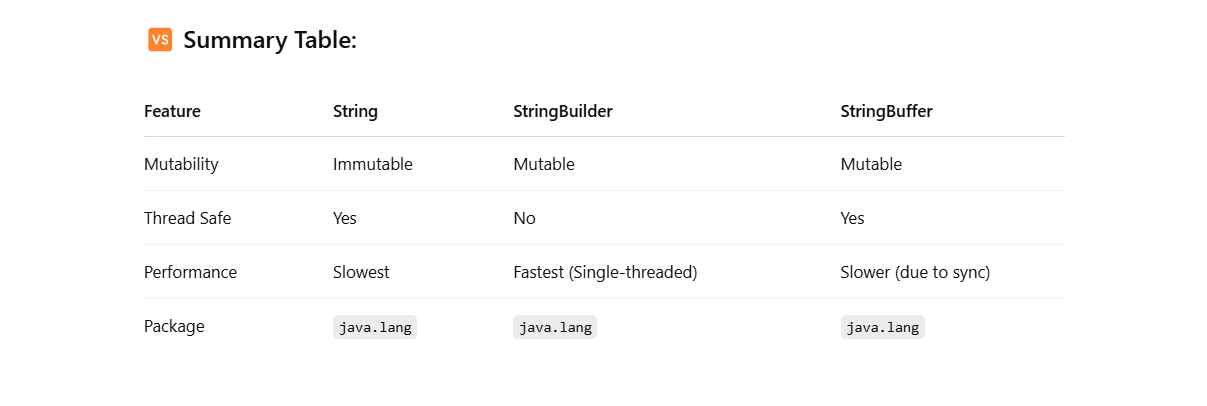
**StringBuffer**

* **Mutable**: Like StringBuilder, can be modified.
* **Thread Safe**: It is synchronized, so it’s safe to use in multi-threaded environments.
* **Performance**: Slower than StringBuilder due to synchronization overhead.
* **Usage**: Use when you need thread safety while modifying strings.

**Eg:**

StringBuffer sbf = new StringBuffer("Hello");

sbf.append(" World"); // Modifies the same object safely in multithreaded context



JAVA INTERNALS:::

**Compiler** – The **javac** tool is used to **compile** Java source code (.java files) into **bytecode** (.class files).

**JVM -**  is the engine that provides a runtime environment which runs the generated byte code which actually executes the program.

The JVM is responsible for memory management (garbage collection), security, and execution of Java programs.

**JRE -**  is a package of software that includes the JVM and other libraries necessary for running Java applications. It's essentially the environment in which Java applications run. It includes

 JVM (which does the execution)

 Core java libraries (such as java.lang, java.util, etc.)

 Other resources needed to run Java applications.

**JDK -**  is a complete development kit for Java developers. Includes everything needed to develop and run Java applications, including the JRE plus development tools (like a compiler, debugger, etc.) to build Java applications.

The **String Pool**, also known as the **String Literal Pool**, is a special memory area in Java that is used to store **string literals**. The String Pool helps **optimize memory usage** by ensuring that only one copy of each literal string is stored in memory, regardless of how many times the string is used in a program.

System – An utility class in java

out– static field of the system class which is object of PrintStream class.

**Garbage Collection:**

In Java, Garbage Collection (GC) is a process by which the Java Virtual Machine (JVM) automatically cleans up memory by deleting objects that are no longer being used by the program**.**

Java handles memory cleanup for you, so you don’t have to manually delete objects like in C or C++.

**But wait, what is memory in Java?**

Whenever you create an object in Java using new, Java allocates some **memory** for that object in a part of memory called the **heap**. For example:

String name = new String("ChatGPT");

Here, a String object is created in memory.

**🧠 Why do we need Garbage Collection?**

Imagine you create hundreds or thousands of objects in your program. Some of those are only needed for a short time. If you dont manually remove them, they just sit there in memory, **taking up space**.

In some languages (like C or C++), **you have to free up memory manually**. But Java does it for you automatically using **Garbage Collection.**

**🔄 How does Garbage Collection work?**

Java keeps track of which objects are being **used (referenced)** and which ones are **not**.

**✅ An object is “reachable” if:**

* It can be accessed from your code.
* There’s a reference pointing to it.

**❌ An object is “unreachable” if:**

* No part of your program can access it anymore.
* It’s like garbage: no one needs it, but it's still lying around.

**Example:**

public class Main {

public static void main(String[] args) {

String s1 = new String("Hello");

s1 = null; // Now "Hello" is unreachable

}

}

When s1 = null; happens, the "Hello" object is now **unreachable**, and Java **marks it for garbage collection**.

**🧹 What exactly does the Garbage Collector do?**

1. It scans the memory (heap).
2. It looks for objects that are **unreachable**.
3. It **deletes them** to free up memory.

**🔧 When does garbage collection happen?**

* You **can’t predict exactly when** it will run.
* The JVM **decides** when it’s the best time.
* You **can request** garbage collection with:

System.gc();

But it’s just a **request**, and JVM might ignore it.

**📦 What’s the Heap ?**

The **heap** is a special region in memory used for dynamic memory allocation where Java stores all objects created during the execution of a program.

* The heap is different from other parts of memory, like the **stack** (which stores local variables and method call information).
* It is **shared among all threads**.
* All **objects** and **arrays** in Java are stored in the heap, no matter their size or complexity.

**How the JVM Manages Heap:**

* **When an object is created**, the JVM automatically decides where to place it in the heap.
* The heap is typically divided into two main regions:
  + **Young Generation** (where new objects are initially allocated).
  + **Old Generation** (where long-lived objects are moved after surviving garbage collection).

**Example Breakdown:**

public class Main {

public static void main(String[] args) {

String str = new String("Hello, Java!");

}

}

1. **Step 1**: new String("Hello, Java!") creates a new object in the heap with the string "Hello, Java!".
2. **Step 2**: The variable str **holds a reference** to that object.
3. **Step 3**: The JVM manages this object in the heap, ensuring it's available for use.

 **Yes**, whenever you create an object in Java using new, the **JVM automatically allocates it in the heap**.

 The **heap** is where all objects live in Java.

 The **garbage collector** takes care of cleaning up objects that are no longer needed, freeing up space in the heap.

**Instance Variables and Their Memory Location:**

**1. In the Heap (Inside the Object)**

* **Instance variables** are stored inside the **object** itself, which is allocated in the **heap**.
* Every time you create an object using the new keyword, a block of memory is allocated in the heap for that object. This block includes the **instance variables** (also called **fields**) that belong to that object.

**Example:**

public class Person {

String name; // instance variable

int age; // instance variable

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person("Alice", 30); // Object created in the heap

}

}

In the above example:

* The Person object is created in the **heap** when you do new Person("Alice", 30).
* Inside the **heap memory** allocated for the person object, the **instance variables** name (a String) and age (an int) are stored.

**Instance Variables in Memory:**

* **Instance variables** are part of the **object** itself, which is stored in the **heap**.
* Each object has its own **set of instance variables**. These variables are **not shared** among different objects of the same class.

Person person1 = new Person("Alice", 30);

Person person2 = new Person("Bob", 25);

person1 and person2 are two different objects in the heap, each with its own copy of the name and age instance variables.

The values "Alice" and 30 are stored in person1, and the values "Bob" and 25 are stored in person2.

**What About the Reference Variables?**

The **reference variables** (like person1 and person2 in the example above) themselves are stored in the **stack**, not the heap.

* A **reference variable** stores the **memory address** of the object in the heap.
* In this case, the **person1 and person2 reference variables** store the **memory location** of the respective Person objects in the heap.

 person1 (reference): stored in **stack**

 new Person("Alice", 30): object stored in **heap**

 "Alice": lives in **string pool** (part of heap)

 **Garbage Collector**: collects unused objects in **heap**, not stack

**Stack Memory(**Not managed by Garbage collector**):**

**What the stack holds:**

* Local variables
* Method parameters
* Return addresses
* Reference variables (that point to heap objects)
* Each thread in a Java program gets its own stack, so if you have multiple threads, each thread will have its own separate stack..
* When a method is called, a **stack frame** is created for that method, and it contains:
  + The **local variables** (like primitive data types or references to objects in the heap).
  + **Method calls** and **return addresses**.

Once the method execution finishes, its stack frame is popped off the stack.

 **Faster** than the heap.

 Objects in the stack **are automatically removed** when they go out of scope (i.e., when the method finishes executing).

 **Primitive types** (like int, double, char) variables are stored in the stack.



 **References** to objects (like String s in the above example) are stored in the stack, but the actual object is stored in the **heap**.

Unlike the stack, the heap is shared across all threads, meaning all threads access the same heap memory.

**🔄 Stack memory is managed automatically by the JVM itself — using a Last In, First Out (LIFO) model.**

Think of it like a pile of plates:

1. When a **method is called**, a new **stack frame** is pushed onto the stack.
2. When the method **returns**, its stack frame is **popped off**, and **all variables inside it are gone** — just like that!

**Example:**

void myMethod() {

int a = 10; // 'a' goes on stack

Person p = new Person("Bob"); // 'p' goes on stack (reference), object goes on heap

}

* When myMethod() is called → a stack frame is pushed.
* a and p live in that frame.
* When myMethod() ends → the stack frame is popped, and both a and p are wiped from memory.

No need for GC — because the JVM **knows** exactly when stack memory is no longer needed.

**🥞 What is LIFO?**

**LIFO (Last In, First Out)** means the **last thing that goes in** is the **first thing to come out**.

Think of a stack of plates:

* You put Plate 1, then Plate 2, then Plate 3.
* To get one, you must take Plate 3 first — the last one you added.

**🧠 In terms of stack memory in Java:**

Every time you call a method, the JVM creates a **stack frame** for that method and **pushes** it on top of the stack.

When the method completes, the stack frame is **popped off** — **last in, first out**.



**🧩 Stack behavior (LIFO):**

**When main is called:**

| main() stack frame | ← pushed first

**Then methodA() is called:**

| methodA() | ← pushed second

| main() |

**Then methodB() is called:**

| methodB() | ← pushed last (top of stack)

| methodA() |

| main() |

**Now methodB() finishes (it's the last in):**

| methodA() | ← methodB() popped (first out)

| main()

**Then methodA() finishes:**

| main() | ← methodA() popped

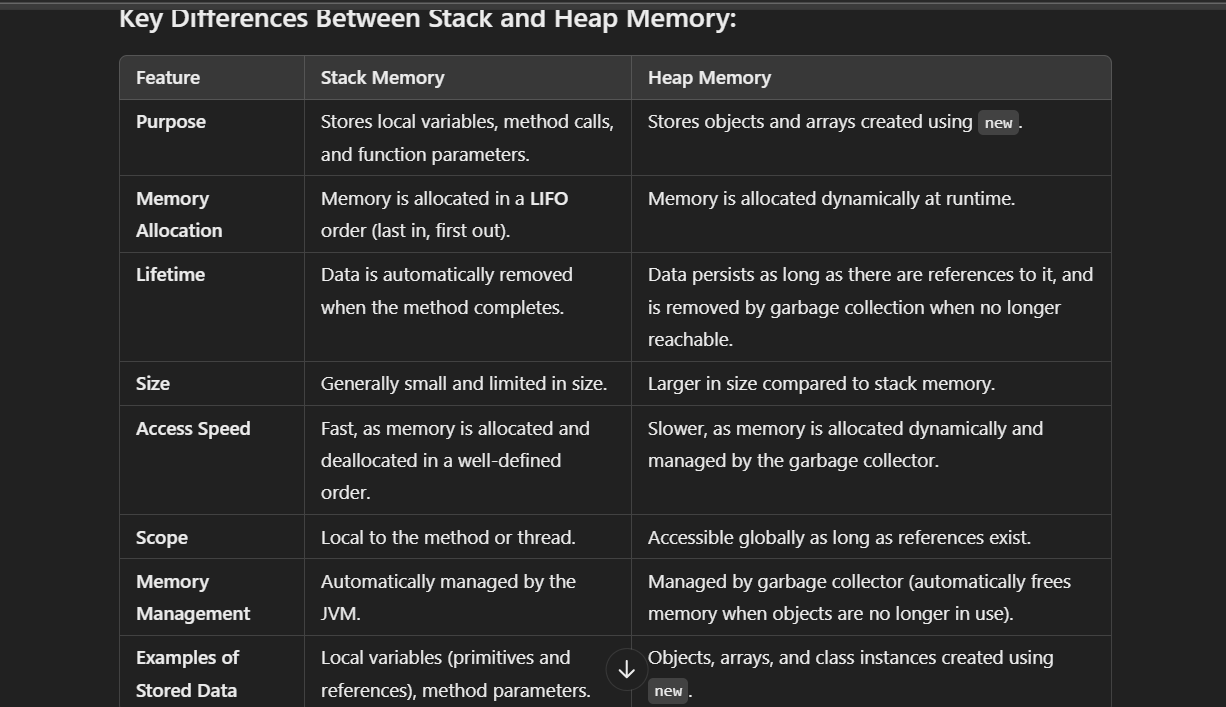
**Then main() finishes:**

(empty stack)

The **last method that was called** is the **first one to finish and get removed** from the stack — exactly what LIFO means.

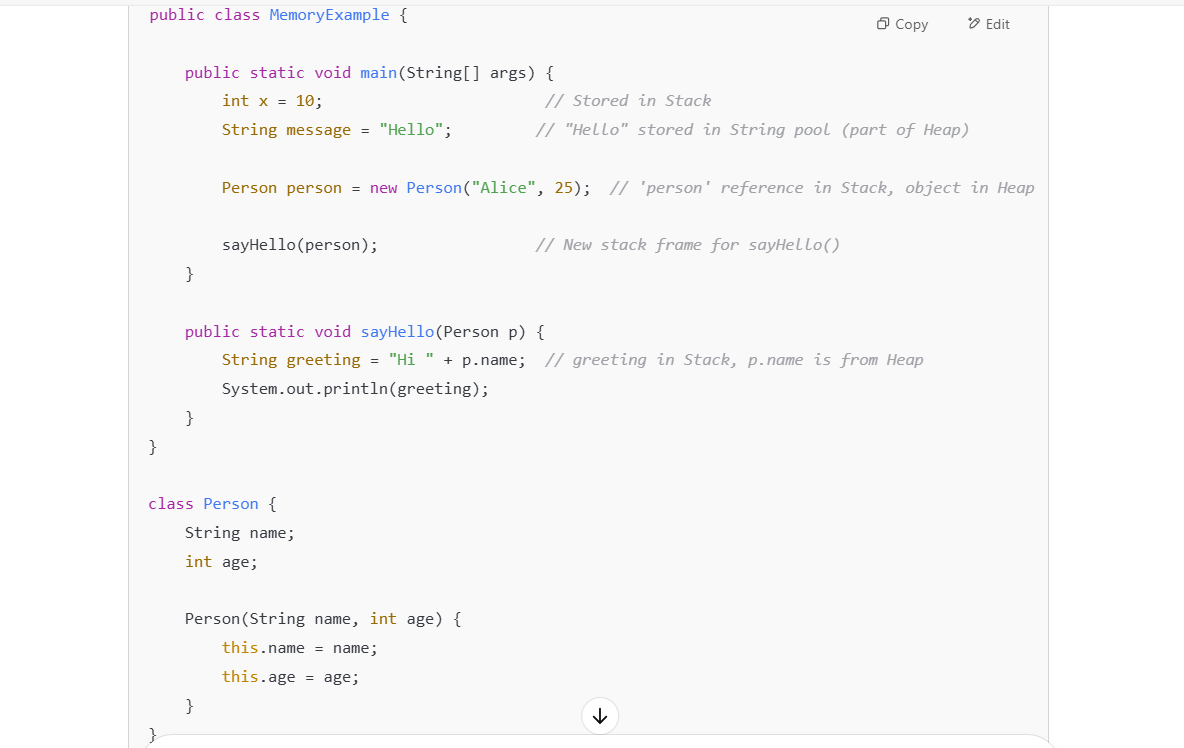
**Quick Recap:**

* **Stack**:
  + Stores **local variables** and **method calls**.
  + Each thread has its own stack.
  + **Fast** and **automatic** memory management.
  + Limited memory and potential risk of **StackOverflowError** with deep recursion.
  + Managed by JVM
* **Heap**:
  + Stores **objects** and **arrays**.
  + Array ‘arrr’ reference stores in stack and actual array stores in heap eg: int[] arr = new int[5];
  + Shared across all threads.
  + **Slower** but more flexible memory management.
  + Managed by JVM’s **Garbage Collection** to free unused memory.



In BRIEF ::

**👩‍💻 Java Code Example**



**🧠 Stack Memory (Thread-local):**

* Method calls (main(), sayHello()) get their own **stack frame**
* Variables like x, message, person (reference), greeting, p and value ‘10’ are stored in the **stack**

**🗄️ Heap Memory (Shared):**

* The Person object **new Person("Alice", 25)** is stored in the **heap**
* The string "**Hello**" which is string literal is stored in the **String Pool** (part of the **heap**)
* The concatenated string "**Hi Alice"** which is new string object is stored in the heap
* Access to heap objects happens through references stored in the **stack**
* name and age are instance variables (belongs to object) stores in heap.

**🔁 What Happens at Runtime:**

1. main() is called → a new stack frame is created
2. Variables x, message, and person reference are pushed to the stack
3. The Person object is created and stored in the heap
4. sayHello() is called → new stack frame created for it
5. Inside sayHello(), the reference p points to the heap Person object

**🧹 Garbage Collection:**

When main() and sayHello() finish:

* Stack frames are popped (deleted)
* If no more references to Person exist, the object is eligible for **garbage collection**

**MAVEN** :

Maven is a build automation and project management tool primarily used for Java-based applications.

It simplifies the process of building, managing dependencies, and automating tasks in Java projects.

* 1. **Build Automation**: It helps automate the compilation, packaging, and deployment of Java applications. With Maven, developers can define how a project should be built in a pom.xml file (Project Object Model), making the process consistent and repeatable.
  2. **Dependency Management**: One of Maven's key features is its ability to manage project dependencies. Instead of manually downloading and configuring third-party libraries, Maven allows you to declare these dependencies in the pom.xml, and it automatically downloads them from a central repository.
  3. **Project Management**: It standardizes the project structure so that developers can follow best practices and conventions. It helps with version control, building consistent releases, and generating documentation.
  4. **Centralized Repositories**: Maven uses central repositories (such as Maven Central) to download and share libraries, plugins, and other resources. This makes it easier to work with external dependencies.
  5. **Plugins and Goals**: Maven uses plugins to perform various tasks such as compiling code, running tests, packaging the application, and deploying it. Plugins are defined in the pom.xml file.

Maven provides a lot of functionality through a set of standard goals and lifecycles, such as:

**Clean**: Removes previous build artifacts.

**Compile**: Compiles the source code.

**Test**: Runs unit tests.

**Package**: Packages the compiled code into a deployable format (e.g., JAR or WAR).

**Install**: Installs the built artifact into the local repository.

**Deploy**: Deploys the artifact to a remote repository.

**Common Maven Commands**

1. **mvn clean**
   * **Goal**: Removes all the files generated by the previous build (like compiled classes, packaged artifacts, etc.).
   * mvn clean will **remove** all the **build-generated files** (like .class files(compiled classes), .jar files, .war files and test results), but it **does not delete your source code** or the project configuration (pom.xml).
   * **Use case**: Before starting a fresh build, you can use clean to ensure that there are no leftovers from previous builds.
   * **Example**: mvn clean
2. **mvn compile**
   * **Goal**: Compiles the source code of the project.
   * **Use case**: This command is used to compile your project's source files in the src/main/java directory and generate the necessary .class files and places them in target/classes directory.
   * **Example**: mvn compile
3. **mvn test**
   * **Goal**: Runs unit tests in your project. (e.g., unit tests written with JUnit).
   * **Use case**: This command runs the test cases (usually written using JUnit or TestNG) and ensures the correctness of your code.
   * **Example**: mvn test
4. **mvn package**
   * **Goal**: Compiles the project, runs the tests, and packages the project into a JAR, WAR, or EAR file (depending on your configuration).
   * **Use case**: This step generates the output artifact that is ready to be deployed or distributed.
   * **Example**: mvn package
5. **mvn install**
   * **Goal**: Runs the full build (compile, test, package) and **installs** the artifact (JAR/WAR/EAR) into your **local Maven repository** (~/.m2/repository)

**What it does**:

* + It **compiles** the project (like mvn compile).
  + It runs any **unit tests** (if you have tests configured in the project).
  + It **packages** the project into a deployable format (like a .jar, .war, or .ear file, depending on your project’s configuration) and puts them in target folder.
  + It **installs** the resulting artifact (like a .jar or .war) into your **local Maven repository** (located in ~/.m2/repository), so that other projects on your machine can use it as a dependency.
  + **Use case**: You use this when you want to **build your project**, **run tests**, **package it**, and **install it into your local Maven repository** so it can be used by other projects.
  + **Example**: mvn install

1. **mvn deploy**
   * **Goal**: deploy the packaged artifact to a remote repository (such as Maven Central or a custom repository).
   * **Use case**: This is typically used when you want to share your artifact with other projects especially in prod environment or make it publicly available.
   * **Example**: mvn deploy

**Summary of Key Maven Commands:**

* **clean**: Cleans the project (removes previous builds) like .class files, .jar, .war files.
* **compile**: Compiles the source code and generates .class files.
* **test**: Runs unit tests.
* **package**: Compiles, runs tests, and packages the code (into .jar, .war, etc.).
* **install**: Runs the full build (compile, test, package) and installs it into the local repository.
* **deploy**: Deploys the project to a **remote repository** (like Maven Central or a private Maven repository).
* **validate**: Validates the configuration and dependencies.
* **site**: Generates documentation for the project.

**Design Patterns::**

Design patterns in Java are standard solutions to common software design problems.

Design patterns are generally divided into **three main categories**:

* Creational
* Structural
* Behavioural

**1. Creational Patterns – How objects are created.**

These patterns deal with object creation mechanisms. Used to define and describe how objects are created at class instantiation time.

Five creational patterns:

1. Singleton
2. Prototype
3. Factory Method
4. Abstract Factory
5. Builder

**Singleton Pattern** – The pattern that restricts the instantiation of a class to one ‘single’ instance.

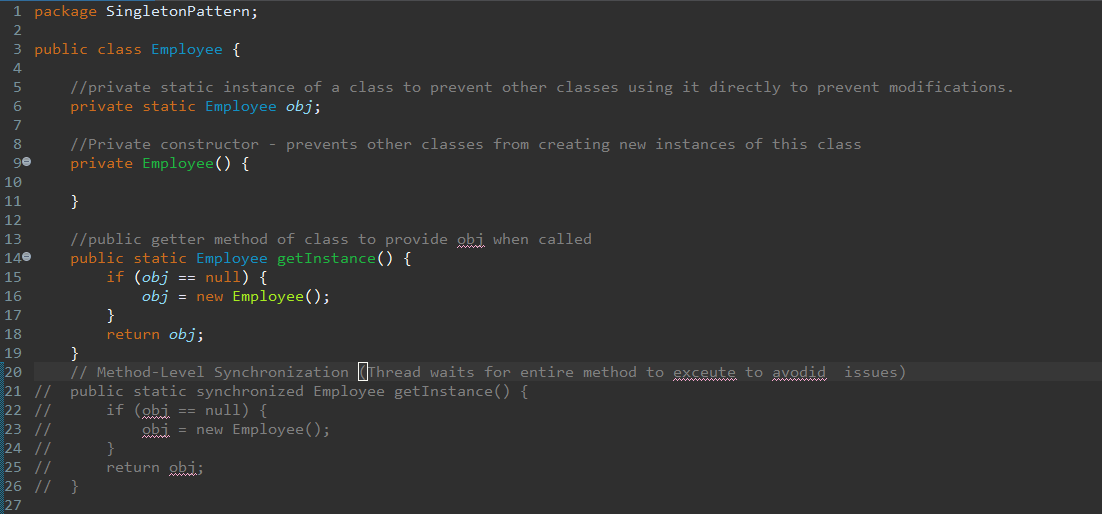
The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

**3 steps to create a singleton class:**

* 1. Create a class with private static instance of a class
  2. Create a private constructor
  3. Create a public static get method such that any class can call it for an object.

**Lazy Singleton(Making it thread safe using synchronized )**

It’s not thread safe if its not eager loading. As multiple threads can access the same method for an object which in turns gives diff objects which violates singleton pattern.



A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

**Eager Singleton(Thread safe by default):**

private static Employee obj = new Employee(); //eager initialization

//eager loading(use with eager initialization)

public static Employee getInstance() {

return obj;

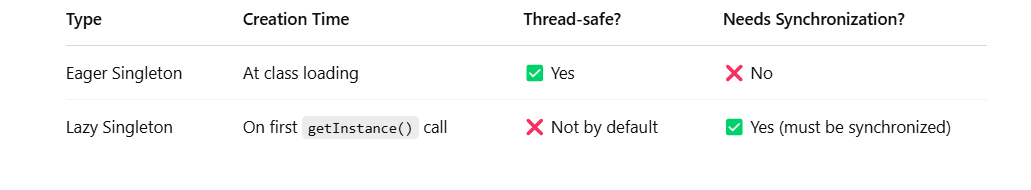
}

**How it works:**

* The Employee obj = new Employee(); line is executed **once** when the class is **loaded** into memory.
* Since it's done at **class loading time**, it's **guaranteed** by the Java Memory Model to be thread-safe.
* getInstance() just returns the same already-created object.
* No matter how many times you call getInstance(), it **always** returns the same obj.
* That’s different from the **lazy singleton**, where the instance is created **only when first requested** — and that’s the one where you can get race conditions if not synchronized.

**Why it's thread-safe without sync**

* Java’s **class loading is thread-safe** by default because its synchronized.
* So obj will be created **once and only once** when class gets loaded, even in a multithreaded environment.
* This makes your code:
  + Simple
  + Fast
  + Thread-safe by default



**Factory Pattern** – The Factory Pattern is called so because it functions similarly to a real-world factory. Just as a factory manufactures products without the client knowing the exact process or details of the manufacturing, the Factory Pattern creates objects **without the client knowing the specific class or instantiation details.**

In the factory pattern, we don’t expose the creation logic to the client and refer the created obj using a standard interface.

**Steps**:

- Create interface and its implementing classes

- Create a main class which calls factory class(factory class creates objs of all classes implementing interface)

- Create a Factory class in a way it creates instances.

- Now client gets required instance from factory class

Client class doesn’t need to know about how obj gets created but gets the objects indirectly using a factory class..so we create a factory class which creates the objects of all classes implementing interfaces and calling the method in factory class to get the required instance.