A close-up of a computer screen

Description automatically generated

A white background with blue text

Description automatically generated

A screenshot of a computer

Description automatically generated

Java:

**What is JVM ? Why is Java called the Platform Independent Programming Language?**

A Java virtual machine (JVM) is a process virtual machine that can execute Java bytecode. Each Java source file is compiled into a bytecode file, which is executed by the JVM.

Java was designed to allow application programs to be built that could be run on any platform, without having to be rewritten or recompiled by the programmer for each separate platform.

A Java virtual machine makes this possible, because it is aware of the specific instruction lengths and other particularities of the underlying hardware platform.

 **JDK**: A full development kit for creating Java applications (includes JRE).

 **JRE**: A runtime environment for executing Java applications (does not include development tools).

**What is the Difference between JDK and JRE ?**

The Java Runtime Environment (JRE) is basically the Java Virtual Machine (JVM) where your Java programs are being executed.

It also includes browser plugins for applet execution.

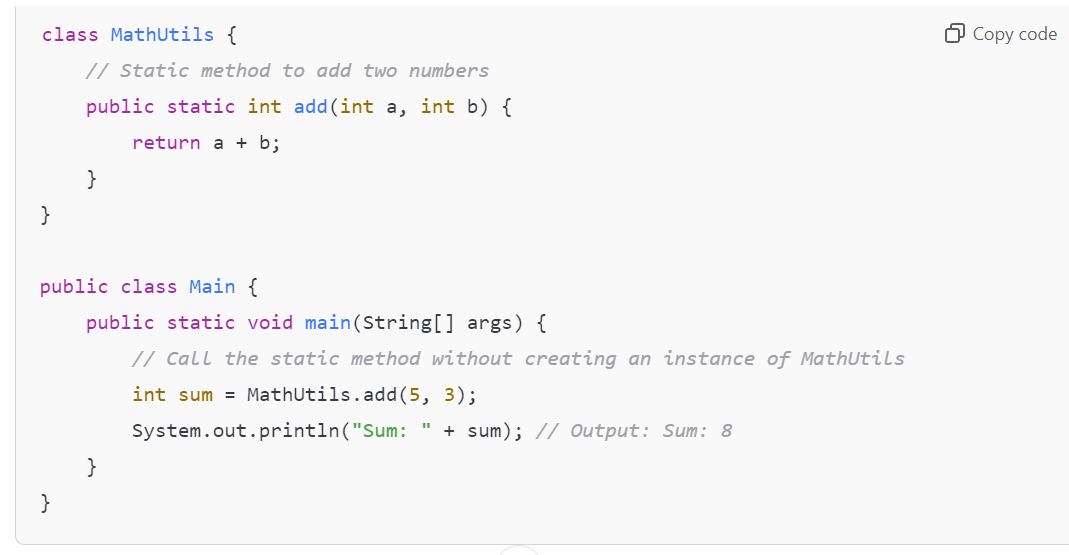
The Java Development Kit (JDK) is the full featured Software Development Kit for Java, including the JRE, the compilers and tools (like JavaDoc, and Java Debugger), in order for a user to develop, compile and execute Java applications

**What does the “static” keyword mean ? Can you override private or static method in Java ?**

**Ans:**

**Static Variables:**

* **Belong to the class rather than any specific instance.**
* **Only one copy exists, shared by all instances of the class.**
* **Can be accessed using the class name without creating an instance**

****

The static keyword denotes that a member variable or method can be accessed, without requiring an instantiation of the class to which it belongs.

A user cannot override static methods in Java, because method overriding is based upon dynamic binding at runtime and static methods are statically binded at compile time.

A static method is not associated with any instance of a classss o the concept is not applicable.

Can you access non static variable in static context ?

Ans: A static variable in Java belongs to its class and its value remains the same for all its instances. A static variable is initialized when the class is loaded by the JVM. If your code tries to access a non-static variable, without any instance, the compiler will complain, because those variables are not created yet and they are not associated with any instance.

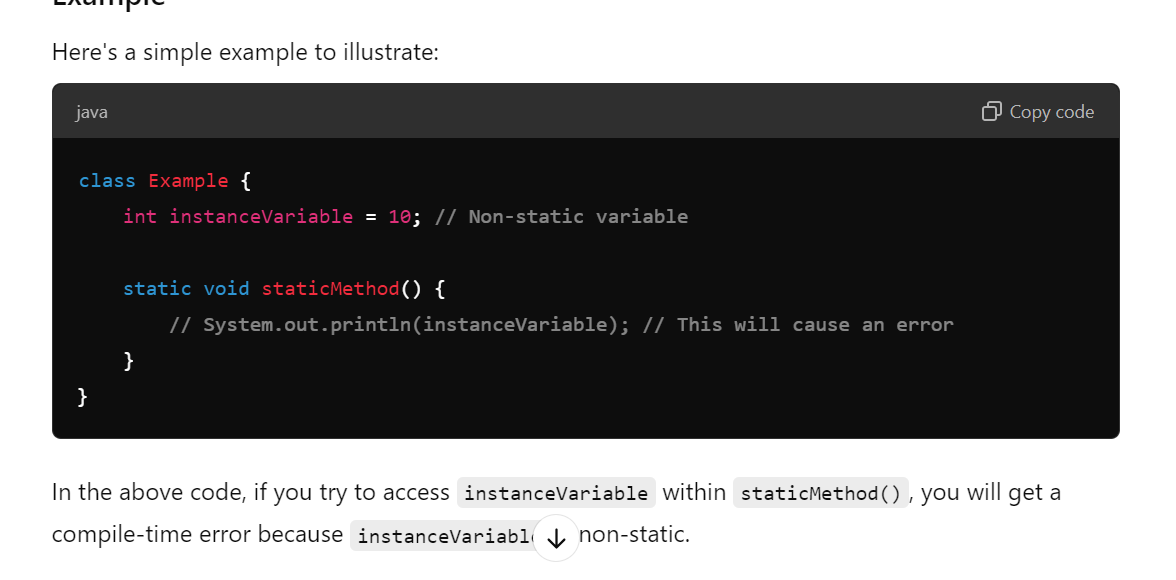
Ans:

No, you cannot directly access a non-static (instance) variable from a static context in Java. Here’s why and how it works:

**Reasons**

1. **Static Context**: A static method or block belongs to the class itself rather than any instance of the class. This means that static members do not have access to instance variables because those variables are tied to specific instances of the class.
2. **No Instance**: Since static methods can be called without creating an instance of the class, there is no guarantee that an instance exists when the static method is executed.

**Example**:



A screenshot of a computer

Description automatically generated----------------------------------------------------------------------------------------

Datatypes:

byte

• short

• int

• long

• float

• double

• boolean

• char

Autoboxing is the automatic conversion made by the Java compiler between the primitive types and their corresponding object wrapper classes. For example, the compiler converts an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this operation is called unboxing.

**What is Function Overriding and Overloading in Java ?**

Method overloading: in Java occurs when two or more methods in the same class have the exact same name, but different parameters.

method overriding: is defined as the case when a child class redefines the same method as a parent class. Overridden methods must have the same name, argument list, and return type. The overriding method may not limit the access of the method it overrides

**What is a Constructor, Constructor Overloading in Java and Copy-Constructor**

A constructor gets invoked when a new object is created. Every class has a constructor.

In case the programmer does not provide a constructor for a class, the Java compiler (Javac) creates a default constructor for that class. The constructor overloading is similar to method overloading in Java. Different constructors can be created for a single class. Each constructor must have its own unique parameter list.

Finally, Java does support copy constructors like C++, but the difference lies in the fact that Java doesn’t create a default copy constructor if you don’t write your own.

**Does Java support multiple inheritance ?**

No, Java does not support multiple inheritance. Each class is able to extend only on one class, but is able to implement more than one interfaces.

to avoid ambiguity and complexity.

Concepts of OOP:

1.Encapsulation :wrapping (binding) of data member and function into single unit is called

Ex:we can make data member private so ther subclass or outside that class nobody will access it.

**Key Benefits of Encapsulation:**

1. **Data Hiding:** Protects the internal state of an object from unintended interference and misuse.
2. **Increased Flexibility and Maintainability:** Changes to the internal implementation can be made without affecting external code that uses the class.
3. **Improved Security:** Sensitive data can be hidden and accessed only through methods

2.Inheritence:that allows one class (the child or subclass) to inherit the properties and behaviors (methods) of another class (the parent or superclass).

**Types of Inheritance in Java**

1. **Single Inheritance**: A class inherits from one superclass.
2. **Multilevel Inheritance**: A class inherits from a superclass, which itself inherits from another class.

**GrandparentClass**

**|**

**|**

**ParentClass**

**|**

**|**

**ChildClass**

1. **Hierarchical Inheritance**: Multiple classes inherit from a single superclass.

**ParentClass**

**/ \**

**/ \**

**ChildClass1 ChildClass2**

1. **Multiple Inheritance**: A class can inherit from multiple classes (not supported directly in Java; achieved through interfaces).

**InterfaceA**

**/ \**

**/ \**

**ClassA ClassB**

**\ /**

**\ /**

**ChildClass**

**3.Polymorphism:**

polymorphism meaning in Java refers to the ability of objects to take on many forms. In other words, it allows different objects to respond to the same message or method call in multiple ways.

Types of Polymorphism

1. Compile-Time Polymorphism (Method Overloading): This is achieved by function overloading, where multiple methods can have the same name with different parameters (different type or number). (method having same name but different parameters)
2. Run-Time Polymorphism (Method Overriding): This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method that gets executed is determined at runtime based on the object being referenced.(method having same name and same parameter)

4.Abstraction: Hiding complex implementation details and showing only the essential features of the object. This can be achieved using abstract classes and interfaces.

**Differences between Abstraction and Encapsulation**

Abstraction and encapsulation are complementary concepts. On the one hand, abstraction focuses on the behavior of an object.

On the other hand, encapsulation focuses on the implementation of an object’s behavior. Encapsulation is usually achieved by

hiding information about the internal state of an object and thus, can be seen as a strategy used in order to provide abstraction.

 **Use Interfaces**: When you need to define a contract for what a class can do, especially when multiple inheritance is needed.

 **Use Abstract Classes**: When you have a common base class that provides shared code and you want to define some common behavior among related classes.

Java Collection Framework:

Collection is set of element

* **List**: An ordered collection that can contain duplicate elements.

**Ex**:ArrayList and LinkedList.

* **Set**: A collection that does not allow duplicates.

**Ex**: HashSet, TreeSet, and LinkedHashSet.

* **Map**: A collection of key-value pairs. Keys are unique, and values can be duplicated.

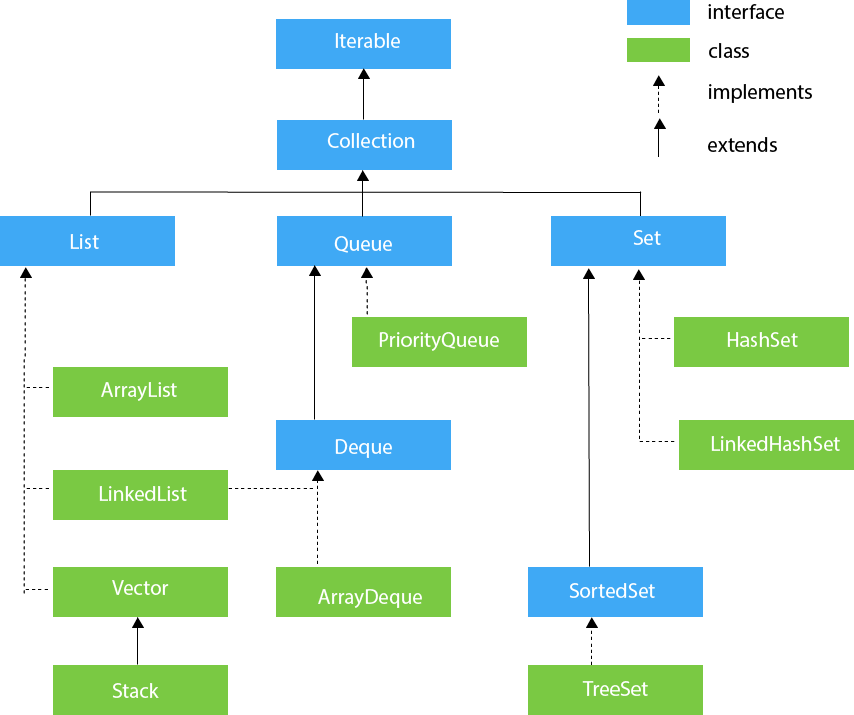
Although Map is part of the **Collection Framework**, it **does not extend** the Collection interface.

Ex:HashMap, TreeMap, and LinkedHashMap.

* **Queue**: A collection designed for holding elements prior to processing. PriorityQueue and LinkedList are common implementations.

A screenshot of a computer screen

Description automatically generated



What makes HashSet an unique collection?

equals() and hashcode() method

What makes TreeSet an unique collection

Compareto method of comparable interface

Iterator:

An **Iterator** in Java is an interface that provides a way to traverse through a collection (like lists, sets, etc.) without exposing the underlying structure of the collection.

**Key Methods of the Iterator Interface**

1. **boolean hasNext()**: Returns true if there are more elements in the collection to iterate over.
2. **E next()**: Returns the next element in the iteration. If no more elements are present, it throws a NoSuchElementException.
3. **void remove()**: Removes the last element returned by the iterator. This method can be called only once per call to next().

**What differences exist between Iterator and ListIterator ?**

The differences of these elements are listed below:

• An Iterator can be used to traverse the Set and List collections, while the ListIterator can be used to iterate only over Lists.

• The Iterator can traverse a collection only in forward direction, while the ListIterator can traverse a List in both directions.

• The ListIterator implements the Iterator interface and contains extra functionality, such as adding an element, replacing an

element, getting the index position for previous and next elements, etc

**How Does a HashMap Work?**

1. **Storing Keys**:
   * When you add a key-value pair to a HashMap, it uses the hashCode() method of the key to figure out where to store that pair. The hashCode() gives a number (the hash code) that indicates the location (or "bucket") in the HashMap where the data should go.
2. **Handling Collisions**:
   * Sometimes, two different keys can produce the same hash code. This is called a "collision."
   * To handle this, the HashMap uses the **equals()** method to check if the keys are actually the same. If two keys have the same hash code, equals() helps determine if they refer to the same key or if they are different keys that just happened to land in the same spot.
   * -----------------

**What is Comparable and Comparator interface ? List their differences.**

* Java provides the Comparable interface, which contains only one method, called **compareTo.**

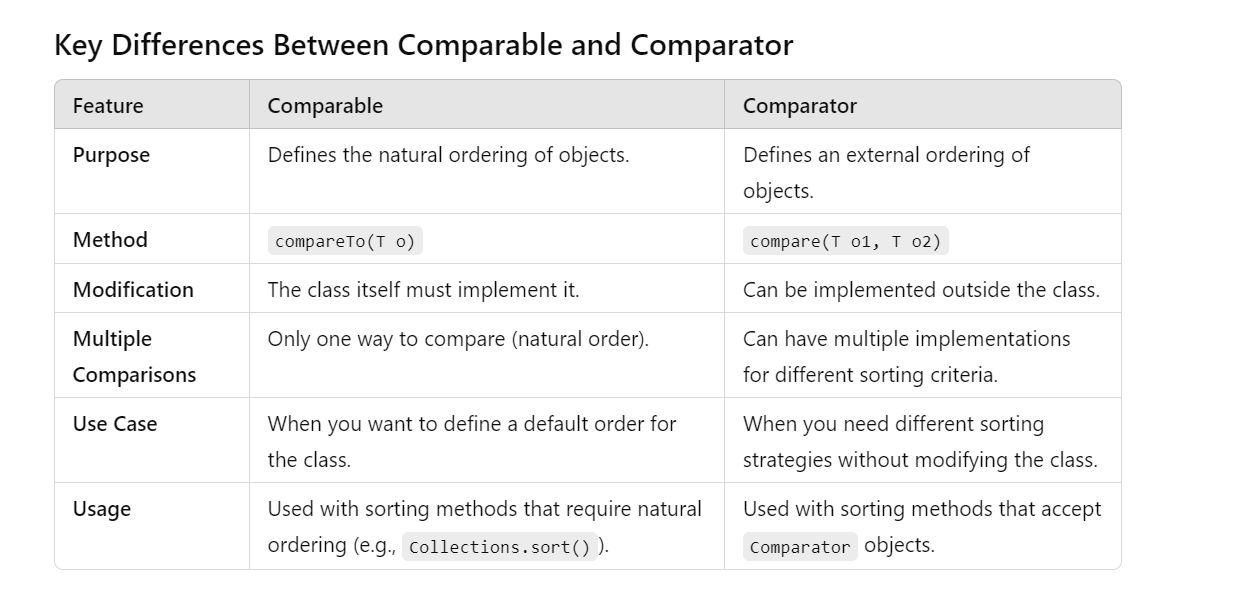
This method compares two objects,in order to impose an order between them. Specifically, it returns a negative integer, zero, or a positive integer to indicate that the input object is less than, equal or greater than the existing object.

* Java provides The Comparator interface, which contains two methods, called **compare and equals**. The first method compares its two input arguments and imposes an order between them.

It returns a negative integer, zero, or a positive integer to indicate that the first argument is less than, equal to, or greater than the second. The second method requires an object as a parameter and aims to decide whether the input object is equal to thecomparator. The method returns true, only if the specified object is also a comparator and it imposes the same ordering as the comparator.

A screenshot of a computer

Description automatically generated



A screenshot of a computer

Description automatically generated

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2)(Insertion,deletion is slower)  Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | Insertion,deletion is faster  Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |
| 5) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |
| 6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized. |
| 7) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list interface. |

**Key Differences Between HashSet and TreeSet**

| **Feature** | **HashSet** | **TreeSet** |
| --- | --- | --- |
| **Implementation** | Uses a hash table | Uses a Red-Black tree (self-balancing binary search tree) |
| **Ordering** | Unordered; no guarantee of order | Ordered; elements are sorted based on their natural ordering or a specified comparator |
| **Time Complexity** | - **Add**: O(1) average case - **Remove**: O(1) average case - **Contains**: O(1) average case | - **Add**: O(log n) - **Remove**: O(log n) - **Contains**: O(log n) |
| **Null Elements** | Allows one null element | Does not allow null elements (throws NullPointerException) |
| **Duplicates** | Does not allow duplicates | Does not allow duplicates |
| **Use Case** | Ideal for fast lookups and when order is not important | Ideal for sorted data and when maintaining order is necessary |
| **Iteration Order** | Iteration order is unpredictable | Iterates in ascending order |
| **Memory Overhead** | Generally uses less memory | Generally has more overhead due to tree structure |

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**What is linked hashmap and its features?**

LinkedHashMap extends HashMap and implements Map

Linked hashmap gurantees order of elements .Elements are retrieved in same order they are inserted.Linked HashMap uses internally double linked lists

to keep insertion order.

The differences between Hashmap and linked hashmap is

1) LinkedHashMap maintains the insertion order while HashMap doesnot maintain order.

2) HashMap if faster for insertion and deletion of elements when compared to linked hashmap. hashmap is preferred only for faster iteration of elements.

**What is concurrent hashmap and its features ?**

Concurrent HashMap is implemented in *java.util.concurrent* package.

Concurrent HashMap extends Abstract Map and implements concurrent Map.

Concurrent HashMap is used in multi threaded environment.

It is similar to Hashtable and synchronized version of hashmap but with minor differences.

Concurrent HashMap *does not allow null keys and values*.

**What differences exist between HashMap and Hashtable ?**

Both the HashMap and Hashtable classes implement the Map interface and thus, have very similar characteristics. However, they

differ in the following features:

• A HashMap allows the existence of null keys and values, while a Hashtable doesn’t allow neither null keys, nor null values.

• A Hashtable is synchronized, while a HashMap is not. Thus, HashMap is preferred in single-threaded environments, while a

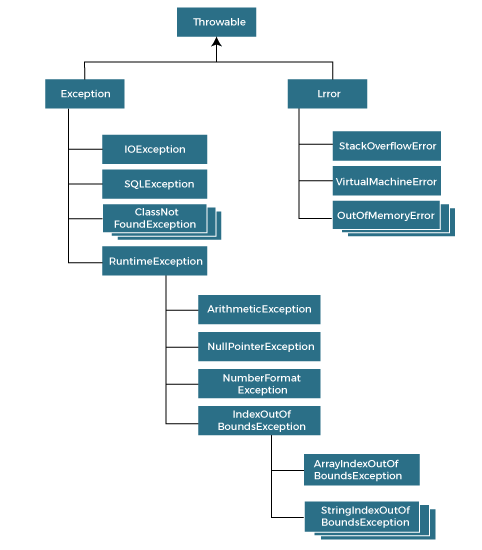
Hashtable is suitable for multi-threaded environments.

• A HashMap provides its set of keys and a Java application can iterate over them. Thus, a HashMap is fail-fast. On the other

hand, a Hashtable provides an Enumeration of its keys.

• The Hashtable class is considered to be a legacy class.

**Exception Handling:**



Types of Java Exceptions



1) Checked Exception (compile time)

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

A screenshot of a computer

Description automatically generated

**throw**

**Definition**: Used to explicitly throw an exception from a method or a block of code.

**Usage**: You can throw both checked and unchecked exceptions

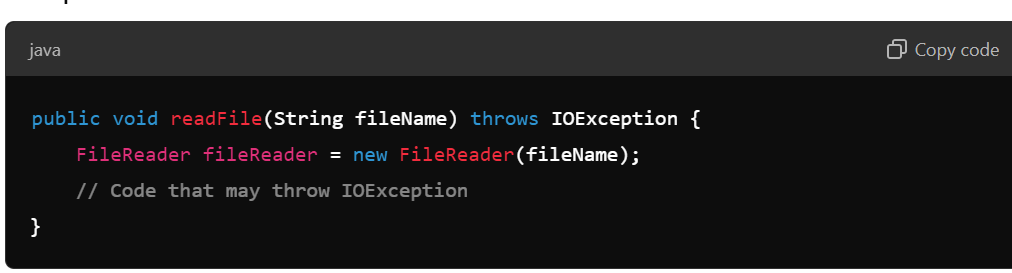
A black screen with white text

Description automatically generated

**throws**

**Definition**: Used in a **method signature to declare that a method can throw one or more exceptions.**

**Usage**: Typically used with checked exceptions that the method does not handle itself. It informs callers that they need to handle these exceptions.



**finally**

* **Definition**: A block of code that follows a try block and is executed after the try block finishes, regardless of whether an exception was thrown or caught.

Commonly used for cleanup activities like closing files or releasing resources.

**Exception Handling Keywords**

1. **try**: The block of code that might throw an exception is enclosed in a try block.
2. **catch**: This block catches and handles the exception. It follows the try block.
3. **finally**: This block is optional and contains code that will always execute, regardless of whether an exception was thrown or caught. It’s commonly used for cleanup operations (e.g., closing files).
4. **throw**: This keyword is used to explicitly throw an exception.
5. **throws**: This keyword is used in method signatures to indicate that a method can throw one or more exceptions.

throw= explicitly throw the exception

throws= method may throw an exception

throwable= superclass of all exception and errors

1. **What is the difference between throw and throws ?**

The throw keyword is used to explicitly raise a exception within the program.

On the contrary, the throws clause is used to indicate those exceptions that are not handled by a method. Each method must explicitly specify which exceptions does not handle, so the callers of that method can guard against possible exceptions. Finally, multiple exceptions are separated by a comma.

 The **throw** statement is used to explicitly throw an exception from a method or a block of code.

 **Usage**: It is followed by an instance of the exception class.

 **Example**: You can use throw when you want to create a specific condition that results in an exception.

A screenshot of a computer program

Description automatically generated

Throws:

The throws keyword is used in a method signature to declare that a method can throw one or more exceptions. It informs callers of the method that they should be prepared to handle these exceptions.

Ex:

A computer screen shot of text

Description automatically generated A screenshot of a computer

Description automatically generated

**What is the importance of finally block in exception handling ?**

A finally block will always be executed, whether or not an exception is actually thrown. Even in the case where the catch statement is missing and an exception is thrown, the finally block will still be executed. Last thing to mention is that the finally

block is used to release resources like I/O buffers, database connections, etc

**What will happen to the Exception object after exception handling ?**

The Exception object will be garbage collected in the next garbage collection

**How does finally block differ from finalize() method ?**

A finally block will be executed whether or not an exception is thrown and is used to release those resources held by the application.

Finalize is a protected method of the Object class, which is called by the Java Virtual Machine (JVM) just before an object is garbage collected.

**Key Differences**

| **Feature** | **finally Block** | **finalize() Method** |
| --- | --- | --- |
| **Purpose** | Execute cleanup code after try-catch | Cleanup before an object is garbage collected |
| **Execution Timing** | Executes after try/catch, regardless of exceptions | Called by the garbage collector, timing is uncertain |
| **Usage Context** | Used in exception handling | Used for resource cleanup before object deletion |
| **Return Type** | Does not return anything | Returns void |
| **Availability** | Can be used in any method | Can only be overridden in classes |

**Conclusion**

* Use the **finally block** to ensure that specific code runs after a try block, regardless of whether an exception occurs.
* Use the **finalize() method** to perform cleanup operations before an object is garbage collected, although its use is discouraged in modern Java due to unpredictability and inefficiency. Instead, consider using try-with-resources for managing resources automatically.

How would you implement a custom exception in Java?

Ans: Implementing a custom exception in Java involves creating a new class that extends either the Exception class for **checked exceptions** or RuntimeException for unchecked exceptions.

**Step-by-Step Implementation**

1. **Create a Custom Exception Class**: Define a new class that extends Exception or RuntimeException.
2. **Define Constructors**: Provide constructors to initialize the exception message and, optionally, the cause of the exception.
3. **Add Any Additional Methods (Optional)**: If necessary, you can add methods to provide more context or functionality related to the exception.

This is Checked Exception

A screen shot of a computer

Description automatically generated

**JDBC:**

**JDBC (Java Database Connectivity)** is a Java API that provides a standard method for connecting to and interacting with databases from Java applications. It allows developers to execute SQL statements, retrieve results, and handle database operations in a platform-independent way.

**Key Features of JDBC:**

1. **Database Independence**: JDBC can connect to any database that has a corresponding JDBC driver, making it versatile for various database management systems (DBMS).
2. **SQL Execution**: It allows you to execute SQL queries, including SELECT, INSERT, UPDATE, and DELETE.
3. **ResultSet Management**: JDBC provides the ResultSet interface for retrieving and manipulating the data returned from a query.
4. **Transaction Management**: JDBC supports transaction management, allowing you to control the commit and rollback of database operations.
5. **Prepared Statements**: It offers the ability to use prepared statements, which

can improve performance and security by preventing SQL injection attacks.

**Basic Steps to Use JDBC:**

1. **Load the JDBC Driver**: The driver is specific to the database you are using (e.g., MySQL, Oracle).
2. **Establish a Connection**: Use the DriverManager class to create a connection to the database.
3. **Create a Statement**: Use the Connection object to create a Statement or PreparedStatement for executing SQL queries.
4. **Execute the Query**: Use the executeQuery() method for SELECT statements and executeUpdate() for INSERT, UPDATE, and DELETE.
5. **Process the ResultSet**: Retrieve data from the ResultSet returned by a SELECT query.
6. **Close the Connection**: Properly close the ResultSet, Statement, and Connection to free up resources.

**Java Streams API**:Use the stream only once

The Java Streams API simplifies data processing by allowing you to perform operations on collections in a functional style. With its support for lazy evaluation, parallel processing, and a rich set of operations, it enables developers to write cleaner and more efficient code. Whether you're filtering, mapping, reducing, or performing aggregate operations, the Streams API provides a powerful toolkit for handling data in Java applications.

: Streams have intermediate operations (like map, filter) that return a new stream and terminal operations (like forEach, collect) that produce a result.

* filter(n -> n % 2 == 0): This intermediate operation filters out all numbers that are not even.
* collect(Collectors.toList()): This terminal operation collects the results into a new list.
* map(n -> n \* n): This intermediate operation applies a function to each element, squaring the numbers.
* reduce(0, Integer::sum): This terminal operation reduces the stream to a single value (the sum), starting with 0
* max(Integer::compareTo): This terminal operation finds the maximum value in the stream.
* orElseThrow(): Ensures that an exception is thrown if the stream is emp

**Functional Interface**:single abstract method

A **functional interface** in Java is an interface that has exactly one abstract method. Functional interfaces can be instantiated with lambda expressions, method references, or constructor references. They are a key feature in Java's support for functional programming introduced in Java 8.

**Key Characteristics of Functional Interfaces:**

1. **Single Abstract Method (SAM)**: A functional interface can have only one abstract method. However, it can have multiple default or static methods.
2. **@FunctionalInterface Annotation**: While it's not mandatory to use this annotation, it helps convey the intent that the interface is meant to be a functional interface. If the interface violates the SAM property, the compiler will generate an error.
3. **Lambda Expressions**: Functional interfaces enable the use of lambda expressions, which allow you to provide the implementation of the abstract method directly in a concise way.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

Abstract class: we cannot create object of abstract class,we can create refrence of abstract class.

* Abstract class contain abstract method and normal method
* It is not mandatory that abstract class having abstract method
* The method define in abstract class i.e abstract method need to implement in child class, if not then make child class also abstract

 **Definition**: An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (methods without a body) as well as concrete methods (methods with an implementation).

 **Purpose**: Abstract classes are used when you want to provide a common base for subclasses. They allow you to define common behavior that can be shared among derived classes.

 **Features**:

* Can contain both abstract methods and concrete methods.
* Can have instance variables (fields).
* Can define constructors.
* Can implement methods from interfaces.
* A class can inherit only one abstract class (single inheritance).

Mutlithreading:

Multithreading is a programming concept that allows concurrent execution of two or more threads, which are lightweight processes.

This can lead to more efficient resource utilization and improved performance, especially in applications that require parallel processing, such as web servers, gaming, or real-time data processing.

**What is the difference between processes and threads ?**

A process is an execution of a program, while a Thread is a single execution sequence within a process.

A process can contain multiple threads. A Thread is sometimes called a lightweight process

**Key Concepts**

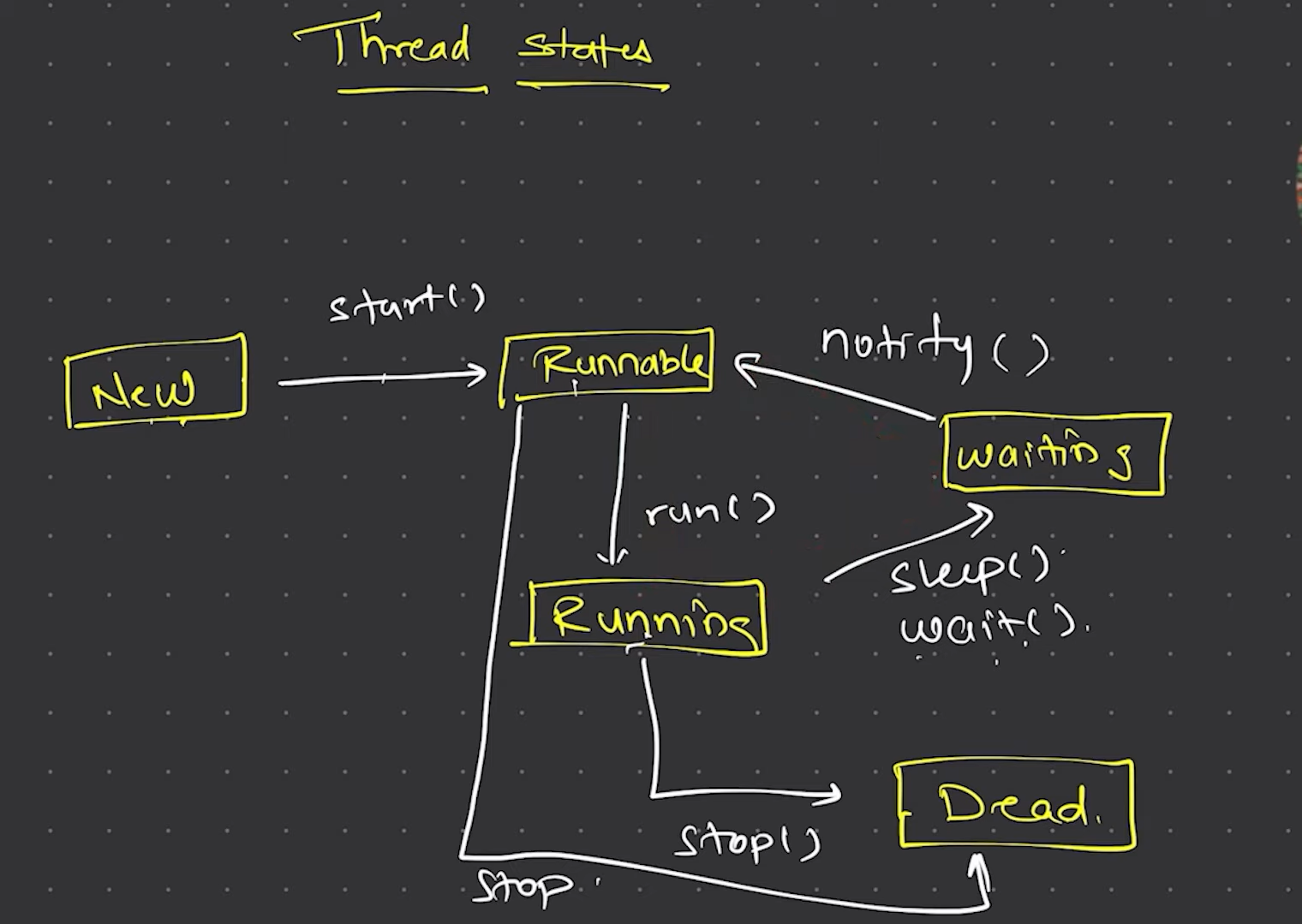
1. **Thread**: A thread is a sequence of instructions that can be managed independently by a scheduler. Threads within the same process share resources but operate independently.
2. **Process vs. Thread**:
   * **Process**: A program in execution, containing its own memory space and resources.
   * **Thread**: A subset of the process; multiple threads share the same memory space and resources of the process.

**Multithreading Benefits**:

**Improved Performance**: Tasks can run concurrently, utilizing CPU resources better.

**Resource Sharing**: Threads can share memory and resources, making communication faster.

**Responsiveness**: Applications can remain responsive (e.g., a UI thread can handle user input while another processes data).



**Thread Lifecycle States**

1. New:
   * A thread is in this state when it is created but not yet started. It can be transitioned to the Runnable state by calling the start() method.
2. Runnable:
   * A thread in this state is ready to run and waiting for CPU time. It can be in the Runnable state after it has been started. This state does not guarantee that the thread is currently executing; it may be waiting in line for CPU allocation.
3. Blocked:
   * A thread enters this state when it is waiting to acquire a monitor lock to enter a synchronized block or method. It cannot proceed until the lock is released.
4. Waiting:
   * A thread is in this state when it is waiting for another thread to perform a specific action (like notifying or interrupting). It can enter this state through methods like wait(), join(), or LockSupport.park().
5. Timed Waiting:
   * Similar to the Waiting state, but the thread is waiting for another thread to perform an action for a specified time. It can enter this state through methods like sleep(milliseconds), wait(milliseconds), or join(milliseconds).
6. Terminated:
   * A thread reaches this state when it has completed execution or has **been terminated. Once a thread is terminated, it cannot be restarted.**
7. **What’s a deadlock ?**

A condition that occurs when two processes are waiting for each other to complete, before proceeding. The result is that both processes wait endlessly

**Explain the available thread states in a high-level.**

During its execution, a thread can reside in one of the following states:

• Runnable: A thread becomes ready to run, but does not necessarily start running immediately.

• Running: The processor is actively executing the thread code.

• Waiting: A thread is in a blocked state waiting for some external processing to finish.

• Sleeping: The thread is forced to sleep.

• Blocked on I/O: Waiting for an I/O operation to complete.

• Blocked on Synchronization: Waiting to acquire a lock.

• Dead: The thread has finished its execution.

**Explain methods of thread and object class**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Explain different ways of creating a thread. Which one would you prefer and**

**why ?**

There are three ways that can be used in order for a Thread to be created:

• A class may extend the Thread class.

• A class may implement the Runnable interface.

• An application can use the Executor framework, in order to create a thread pool.

The Runnable interface is preferred, as it does not require an object to inherit the Thread class. In case your application design requires multiple inheritance, only interfaces can help you. Also, the thread pool is very efficient and can be implemented and

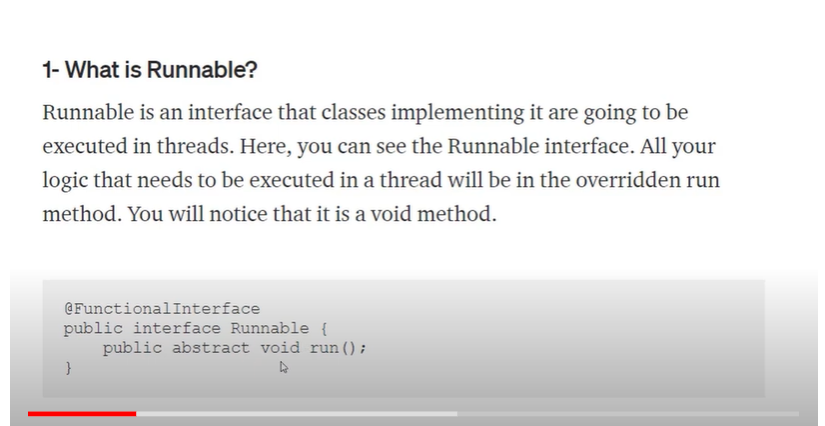
used very easily.

**What is the difference between a synchronized method and a synchronized**

**block ?**

In Java programming, each object has a lock. A thread can acquire the lock for an object by using the synchronized keyword.

The synchronized keyword can be applied in a method level (coarse grained lock) or block level of code (fine grained lock).



**Runnable**

**Definition**: An interface **representing a task that can be executed by a thread**.

**Method**:

* + void run(): This method contains the code that defines the task to be executed.

**Return Type**: Does not return a result; the return type **is void.**

**Exception Handling: Cannot throw** checked exceptions directly.

**Usage**: Commonly used for tasks that **do not need to return a value.**

**A screen shot of a computer program

Description automatically generated**

**Callable**

**Definition:** An interface that represents a task that can **be executed by a thread and returns a result.**

**Method:**

* + V call(): This method contains the code that defines the task and returns a value of type V.

**Return Type:** Can return a result; the return type is a generic type <V>.

**Exception Handling:** Can throw checked exceptions, allowing for more robust error handling.

**Usage:** Ideal for tasks that need to return a result or throw exceptions.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

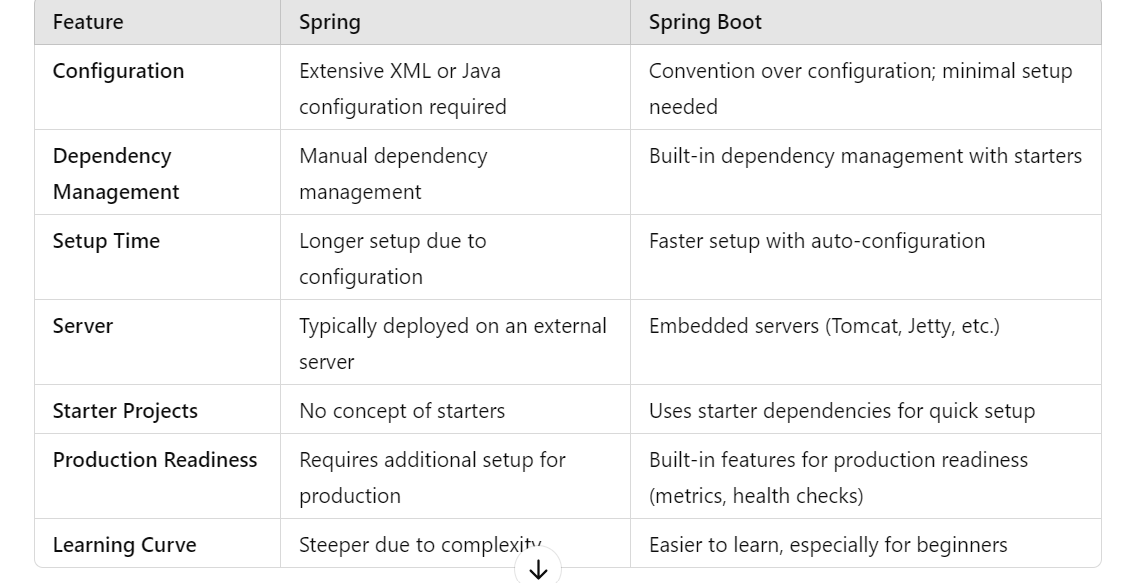
Description automatically generated

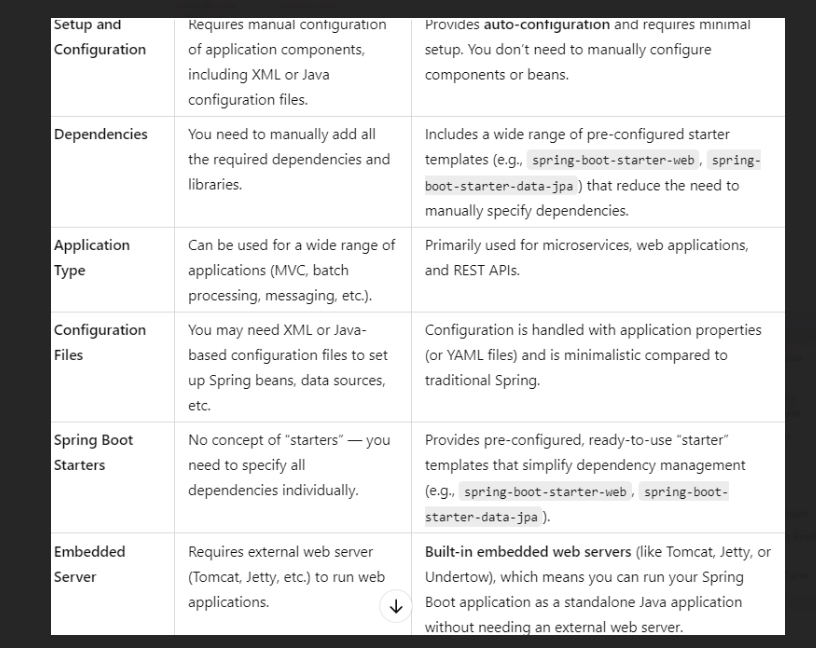
A screen shot of a computer

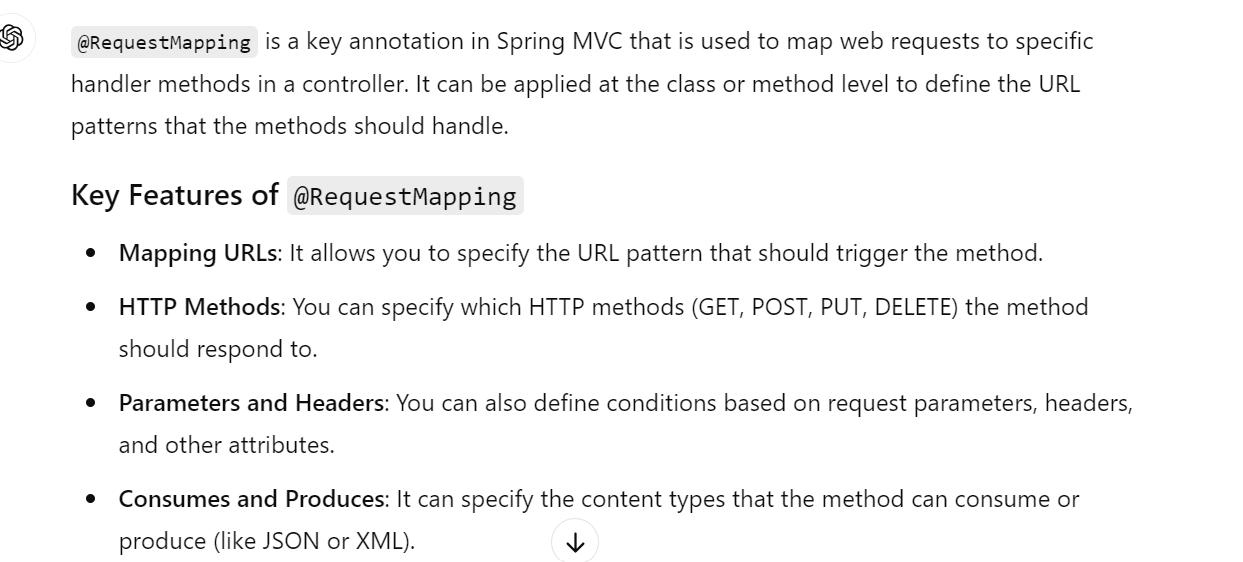
Description automatically generated

A screen shot of a computer

Description automatically generated



--------------





A screenshot of a computer program

Description automatically generated

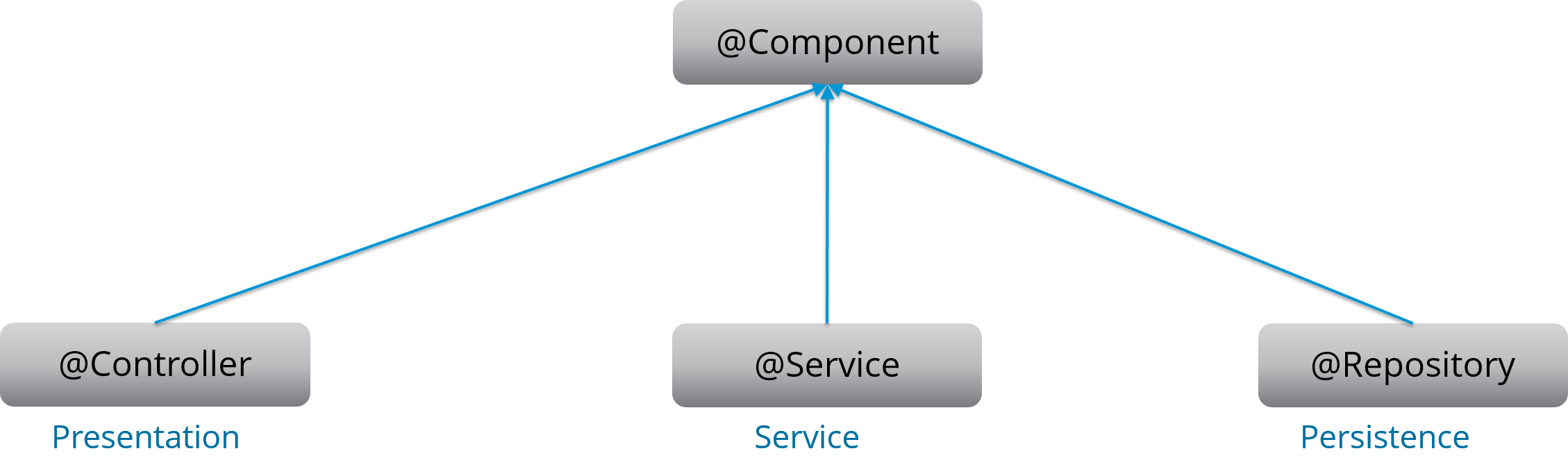
A screenshot of a computer

Description automatically generated

Springboot annotation:

A screenshot of a chat

Description automatically generated

****

A screenshot of a chat

Description automatically generated

1. Controller vs Restcontroller

• @Controller is used when you want to **render views** (HTML, JSP, etc.) and return a **view name**.

•  @RestController (combination of responsebody and controller)is used when you want to **return data** (JSON, XML) directly, typically for building **REST APIs**.

A screenshot of a computer program

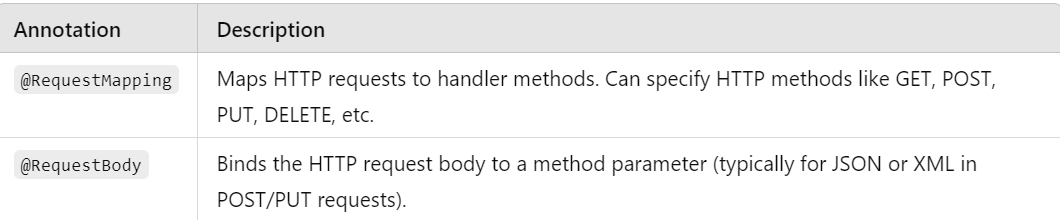
Description automatically generated

1. Difference between spring and Springboot

A screenshot of a computer program

Description automatically generated

1. @RequestBody and @RequestMapping



A white background with black text

Description automatically generated

1. Http vs Https

• **HTTP** is fast but insecure. It is suitable for non-sensitive, public information like blogs, news sites, or marketing pages.

•  **HTTPS** is **more secure** and should be used for sites handling sensitive data, such as login pages, financial transactions, or online stores, as it ensures encryption, integrity, and trust.

A screenshot of a computer

Description automatically generated

1. Spring Annotations

@Component: to specify the class as spring managed bean. When @Component is used it automatically detect and register as a bean in the application context during component scanning.

@Service:

**Purpose**: Indicates that a class provides a **service**.

**Use**: It’s a specialization of @Component, typically used in the service layer of an application to perform business logic.

**@Repository**

**Purpose**: Indicates that a class is a **DAO (Data Access Object)** responsible for interacting with the database.

**Use**: It is a specialization of @Component

**@Controller**

**Purpose**: Marks a class as a **Spring MVC controller**.

**Use**: It is used to handle HTTP requests in a Spring web application (typically in MVC applications).

javaA screenshot of a computer

Description automatically generated

**@RestController**

**Purpose**: Combines @Controller and @ResponseBody.

**Use**: Used for **RESTful web services**. It automatically converts the return value of methods into HTTP responses (usually JSON or XML), and you don’t need to annotate each method with @ResponseBody.

**@Autowired**

**Purpose**: Performs **dependency injection** by automatically injecting a bean into a class.

**Use**: It can be used on **fields**, **constructors**, or **setter methods** to inject the required dependencies automatically.

**@Value**

**Purpose**: Injects **property values** into Spring beans.

**Use**: You can use @Value to inject values from properties files (like application.properties or application.yml) or environment variables into a bean.

**@Configuration**

**Purpose**: Marks a class as a **configuration** class that provides **bean definitions**.

**Use**: It is a Java-based alternative to XML configuration files. It is used to define beans using the @Bean annotation inside the class.

**@Bean**

**Purpose**: Indicates that a method will **return a bean** to be managed by the Spring container.

**Use**: Typically used inside a @Configuration class to define beans explicitly.

**@RequestMapping**

**Purpose**: Maps **HTTP requests** to handler methods of MVC and REST controllers.

**Use**: It can be used to handle different HTTP methods like GET, POST, PUT, DELETE, etc., and map them to methods in a controller.

**@GetMapping, @PostMapping, @PutMapping, @DeleteMapping**

**Purpose**: Simplified versions of @RequestMapping for specific HTTP methods.

**Use**: These annotations are more concise alternatives to @RequestMapping when handling specific HTTP methods like GET, POST, PUT, DELETE, etc.

**@Scope**

**Purpose**: Specifies the **scope** of a Spring bean.

**Use**: Defines how long a bean lives and how it is shared in the Spring container. Common scopes are singleton (one instance for the application) and prototype (new instance every time).

**@PostConstruct and @PreDestroy**

**Purpose**: These annotations define **callback methods** that should be executed after the bean is initialized and before it is destroyed, respectively.

**Use**: @PostConstruct is called after the bean has been instantiated, while @PreDestroy is called just before the bean is destroyed.

**@Transactional**

**Purpose**: Marks a method or class to be **transactional**.

**Use**: It ensures that the method runs within a **transaction**. If an exception occurs, the transaction is rolled back, and no changes are committed to the database.

**@SpringBootApplication**

**Purpose**: This is a convenience annotation that encapsulates three important annotations **(@Configuration, @EnableAutoConfiguration, and @ComponentScan)** used in a Spring Boot application. It marks the main class of a Spring Boot application.

**Use**: It simplifies the setup of a Spring Boot application by automatically enabling component scanning and auto-configuration. It's typically placed on the main class that contains the public static void main(String[] args) method.

**@Qualifier**

**Purpose**: Used in combination with @Autowired, it helps resolve ambiguity when multiple beans of the same type are present in the Spring context. It specifies which bean to inject when there are multiple candidates.

**Use**: It's used to differentiate between multiple beans of the same type.

**@Inject**

**Purpose**: Part of the javax.inject package, this annotation is used for dependency injection, similar to @Autowired. It is a more standard annotation from JSR-330 (Java Dependency Injection).

**Use**: It can be used to inject dependencies into a Spring-managed bean, just like @Autowired, but without Spring-specific ties.

**Usage**: @Inject can be used in place of @Autowired for dependency injection if you're using JSR-330 standards or prefer a non-Spring-specific annotation.

**@ComponentScan**

**Purpose**: Tells Spring where to scan for **Spring components** (i.e., classes annotated with @Component, @Service, @Repository, @Controller, etc.).

**Use**: By default, Spring Boot uses @SpringBootApplication, which includes @ComponentScan. If you need to customize or specify a package to scan for Spring components explicitly, you can use @ComponentScan.

**@PathVariable**

**Purpose**: Used to extract values from the URI (path) of a URL. This is often used in RESTful APIs to retrieve values passed as part of the path.

**Use**: It's used in controller methods to access path parameters from the URL.

**Usage**: Use @PathVariable to extract values from the URL. For instance, in /users/{id}, the value of {id} will be extracted and passed to the method as a parameter.

**@RequestParam**

**Purpose**: Used to extract query parameters from the URL (e.g., in a GET request). These parameters are usually passed as part of the query string in the URL.

**Use**: It's used to retrieve request parameters in controller methods when dealing with form submissions or URL query parameters.

What is the difference between == and equals() in Java?

**1. == Operator**

* **Type**: Reference comparison (for objects) or value comparison (for primitives).
* **Usage**:
  + When comparing primitive data types (e.g., int, char), == checks for value equality.
  + When comparing objects, == checks if two references point to the same object in memory (i.e., reference equality).

A screenshot of a computer program

Description automatically generated

**2.equals() Method**

**Type**: Method for checking logical equality.

**Usage**:

* + The .equals() method is intended to compare the actual contents or state of objects.
  + The default implementation of .equals() in the Object class checks for reference equality (same as ==), but it can be overridden in derived classes to provide specific equality logic.



A computer screen shot of a program

Description automatically generated

Explain the concept of immutability and how it applies to Strings.

Ans:

**Immutability** refers to the property of an object whose state cannot be modified after it is created. In other words, once an immutable object is instantiated, its data cannot be changed. Instead of modifying the object itself, any operation that seems to alter its state will instead create a new instance of the object with the updated values.

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

**Benefits of Immutability**

1. **Thread Safety**: Immutable objects are inherently thread-safe, meaning multiple threads can access them without the risk of inconsistency or corruption since their state cannot be altered.
2. **Simplicity**: They are easier to understand and reason about because their state is constant throughout their lifetime.
3. **Caching and Performance**: Since immutable objects can be shared freely, they can be cached and reused, which can improve performance and reduce memory overhead.
4. **Security**: Immutable objects provide a level of security as their state cannot be changed, making them less vulnerable to unintentional side effects.

**String Pool**: Java maintains a special memory area called the **String Pool**. When you create a String using a string literal, Java checks the pool first to see if an equivalent String already exists. If it does, it returns a reference to that object instead of creating a new one. This is why == can sometimes return true for strings with the same content:

A black screen with white text

Description automatically generated

**Method References in Java**

**Method references** are a shorthand notation for referring to methods without invoking them, commonly used with functional interfaces.

They simplify code by reducing boilerplate and enhancing readability.

**Types of Method References**

1. **Static Method**: ClassName::staticMethodName
   * Example: Function<Integer, Integer> square = MathUtils::square;

A screenshot of a computer

Description automatically generated

1. **Instance Method of a Particular Object**: instance::instanceMethodName
   * Example: Runnable runnable = greeting::sayHello;

A screenshot of a computer program

Description automatically generated

1. **Instance Method of an Arbitrary Object**: ClassName::instanceMethodName
   * Example: names.sort(String::compareTo);

A black screen with white text

Description automatically generated

1. **Constructor Reference**: ClassName::new
   * Example: Supplier<Person> personSupplier = Person::new;

A screenshot of a computer screen

Description automatically generated

**Benefits**

* **Conciseness**: Reduces the need for verbose lambda expressions.
  + Example: names.forEach(System.out::println);
* **Clarity**: Makes code easier to read and understand.
* **Reduces Anonymous Classes**: Leads to cleaner code.
* **Supports Functional Style**: Fits well with Java's functional programming

features.

How does the Optional class help in avoiding NullPointerExceptions?

The Optional class in Java is a container object that may or may not contain a non-null value. Introduced in Java 8, it helps to avoid NullPointerExceptions (NPEs) by providing a more expressive way to handle the absence of a value without using null references.

**Advantages of Spring Boot’s Auto-Configuration**

**Spring Boot's auto-configuration is a powerful feature that simplifies the setup of Spring applications by automatically configuring beans based on the dependencies present in the classpath**

1. **Reduced Configuration Overhead**: Minimizes boilerplate code with sensible defaults, promoting a convention-over-configuration approach.
2. **Rapid Development**: Allows quick setup, enabling developers to focus on business logic rather than configuration.
3. **Automatic Bean Registration**: Conditionally configures beans based on the classpath, ensuring only relevant components are created.
4. **Seamless Integration**: Easily integrates with various Spring projects (e.g., Spring Security, Spring MVC) with minimal effort.
5. **Customization and Override**: Provides easy ways to customize configurations using properties files or by defining your own beans.
6. **Built-in Metrics and Monitoring**: Automatically configures monitoring endpoints via the actuator module for easy application management.
7. **Testing Support**: Simplifies test setups, allowing developers to focus on writing tests instead of boilerplate.
8. **Community-Driven**: Continuously updated by the active Spring community, ensuring extensibility and support for various libraries.

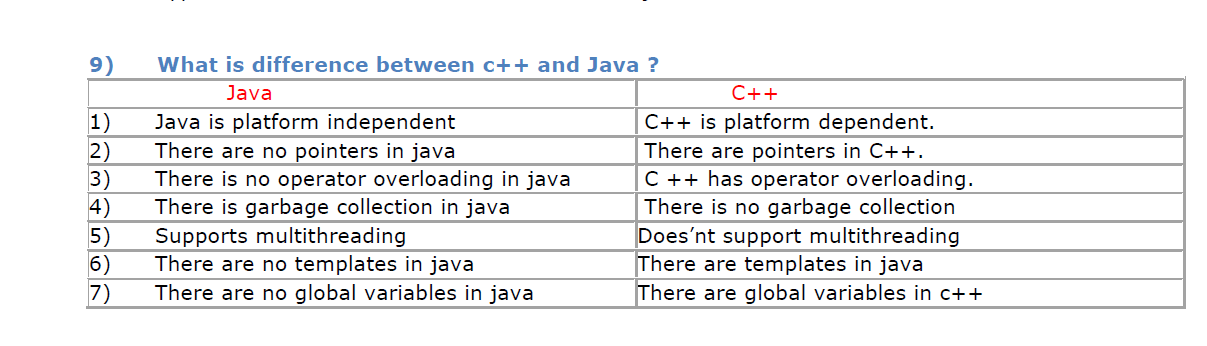
**How to call one constructor from the other constructor ?**

With in the same class if we want to call one constructor from other we use this() method. Based on the number of parameters we pass appropriate this() method is called.

Restrictions for using this method :

1) this must be the first statement in the constructor

2)we cannot use two this() methods in the constructor



**What is ‘IS-A ‘ relationship in java?**

‘is a’ relationship is also known as inheritance. We can implement ‘is a’ relationship or inheritance in java

using extends keyword. The advantage or inheritance or is a relationship is reusability of code instead of

duplicating the code.

Ex : Motor cycle is a vehicle

Car is a vehicle Both car and motorcycle extends vehicle.

**What is ‘HAS A’’ relationship in java?**

‘Has a ‘ relationship is also known as “composition or Aggregation”. As in inheritance we have ‘extends’

keyword we don’t have any keyword to implement ‘Has a’ relationship in java. The main advantage of

‘Has-A‘ relationship in java code reusability.

E

**Explain the importance of throwable class and its methods?**

Throwable class is the root class for Exceptions. All exceptions are derived from this throwable class. The

two main subclasses of Throwable are Exception and Error. The three methods defined in throwable class

are :

1) void printStackTrace() :

This prints the exception information in the following format :

Name of the exception, description followed by stack trace.

2) getMessage()

This method prints only the description of Exception.

3) toString():

It prints the name and description of Exception.

**Explain when ClassNotFoundException will be raised ?**

When JVM tries to load a class by its string name, and couldn’t able to find the class

classNotFoundException will be thrown. An example for this exception is when class name is misspelled

and when we try to load the class by string name hence class cannot be found which raises

ClassNotFoundException

Stream Api Ex:

import java.util.List;  
import java.util.stream.Collectors;  
import java.util.Arrays;

public class StreamsExample {  
    public static void main(String[] args) {  
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Using streams to filter and perform actions on a collection  
        List<Integer> evenNumbers = numbers.stream()  
                                           .filter(n -> n % 2 == 0)  
                                           .collect(Collectors.toList());  
  
        System.out.println("Even Numbers: " + evenNumbers);  
  
        // Using map to square each number  
        List<Integer> squaredNumbers = numbers.stream()  
                                              .map(n -> n \* n)  
                                              .collect(Collectors.toList());  
  
        System.out.println("Squared Numbers: " + squaredNumbers);  
    }  
}

A functional interface in Java is an interface that contains exactly one abstract method. These interfaces can be instantiated using lambda expressions, method references, or anonymous classes. Functional interfaces are a key feature of Java's support for functional programming introduced in Java 8.

**Key Characteristics**

1. **Single Abstract Method**: A functional interface must have exactly one abstract method. It can have multiple default or static methods, but only one abstract method.
2. **@FunctionalInterface Annotation**: While it's not mandatory, it's good practice to use the @FunctionalInterface annotation. This annotation helps the compiler enforce that the interface is indeed a functional interface.
3. **Usage in Lambda Expressions**: Functional interfaces provide the target type for lambda expressions and method references, making them essential for writing concise and expressive code.

 --------------------------------

* + It works seamlessly with various JPA providers like Hibernate, allowing you to use the full capabilities of JPA.

what is the significance of @springbootapplication

The @SpringBootApplication annotation in Spring Boot is a convenient way to enable several key features and configurations in a Spring application. It serves as a combination of three important annotations, which makes it easier to bootstrap and configure your Spring application. Here’s a breakdown of its significance:

**Components of @SpringBootApplication**

1. **@Configuration**: This indicates that the class can be used by the Spring IoC (Inversion of Control) container as a source of bean definitions. You can define beans within this class using methods annotated with @Bean.
2. **@EnableAutoConfiguration**: This enables Spring Boot’s auto-configuration feature, which automatically configures your application based on the dependencies present on the classpath. For example, if you have spring-boot-starter-web in your project, it will automatically configure Spring MVC and an embedded web server.
3. **@ComponentScan**: This tells Spring to scan the package where the application is located for other components, configurations, and services. This allows you to use @Component, @Service, @Repository, and other annotations without having to specify explicit bean definitions.

**Significance**

1. **Simplified Configuration**: By combining multiple annotations into one, @SpringBootApplication significantly reduces the amount of boilerplate code needed to set up a Spring application.
2. **Automatic Setup**: The auto-configuration feature helps developers quickly set up an application with sensible defaults, which can be overridden by custom configurations when needed.
3. **Component Scanning**: It ensures that all your Spring components are detected automatically, making it easier to organize your code without manual configuration.
4. **Single Entry Point**: The class annotated with @SpringBootApplication is usually the main entry point of your Spring Boot application, making it clear where the application starts.

Spring Boot Starter:

* Spring boot starters are a set of conveninent dependency descriptions that u can include in your application.
* They aggregate common dependencies for a specific functionality,reducing the complexity of managing versions and comaptibility.

**Examples of Common Starters:**

1. **spring-boot-starter-web**: For building web apps (includes Spring MVC and Tomcat).
2. **spring-boot-starter-data-jpa**: For database access using JPA (includes Hibernate).
3. **spring-boot-starter-security**: For adding security features (authentication and authorization).
4. **spring-boot-starter-test**: For testing your application (includes JUnit and Mockito).

what is the use of @value

is used to inject values in to filelds from property files ,environment variables,or system properties.

A black rectangular object with white text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A black and white box with white text

Description automatically generated

What is Spring Boot Actuator?

Ans:It provides build in endpoints for monitoring and managing your application.

It allows you to access applications metrics,health checks,enviornment properties and mores,

what testing frameworks are commonly used with spring boot?

Here’s a concise overview of commonly used testing frameworks with Spring Boot:

**1. JUnit**

* **Description**: The primary framework for unit and integration testing in Java.
* **Usage**: Basic test structure and context loading.

**2. Mockito**

* **Description**: A mocking framework for creating mock objects.
* **Usage**: Mock dependencies in unit tests.

**3. Spring Test**

* **Description**: Part of the Spring Framework for testing Spring components.
* **Usage**: Supports integration tests with the Spring context.

**4. AssertJ**

* **Description**: A fluent assertion library for readable tests.
* **Usage**: Enhances assertions in your tests.

**5. JUnit 5 (Jupiter)**

* **Description**: The next generation of JUnit with advanced features.
* **Usage**: Supports parameterized tests and improved annotations.

what are common deployment options for spring boot applications

Ans:

* **Standalone Deployment**: Run as a standalone JAR using java -jar your-app.jar.
* **Docker**: Package the application in a Docker container for easy deployment across different environments.
* **Cloud Platforms**: Deploy to services like AWS Elastic Beanstalk, Google App Engine, or Azure Spring Apps for scalable cloud solutions.
* **Kubernetes**: Use Kubernetes for managing containerized applications with features like scaling and load balancing.
* **Heroku**: Deploy easily to Heroku by pushing your code, which handles the rest.
* **Traditional Application Servers**: Deploy as a WAR file on servers like Tomcat or WildFly.
* **Serverless Platforms**: Use AWS Lambda or similar services for a serverless architecture, deploying as functions.

Types of locks in java

**Summary of Locks in Java:**

* **synchronized keyword**: Simple and easy to use for method or block-level synchronization. Ideal for basic use cases.
* **Lock interface (ReentrantLock)**: More flexible with features like tryLock(), timed locks, and manual unlocking.
* **ReadWriteLock**: Allows multiple readers to access a resource concurrently, but only one writer.
* **StampedLock**: Optimized for performance with additional read and write modes, and optimistic reads. Introduced in java8

**Design Patterns:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

SOLID PRINCIPLES

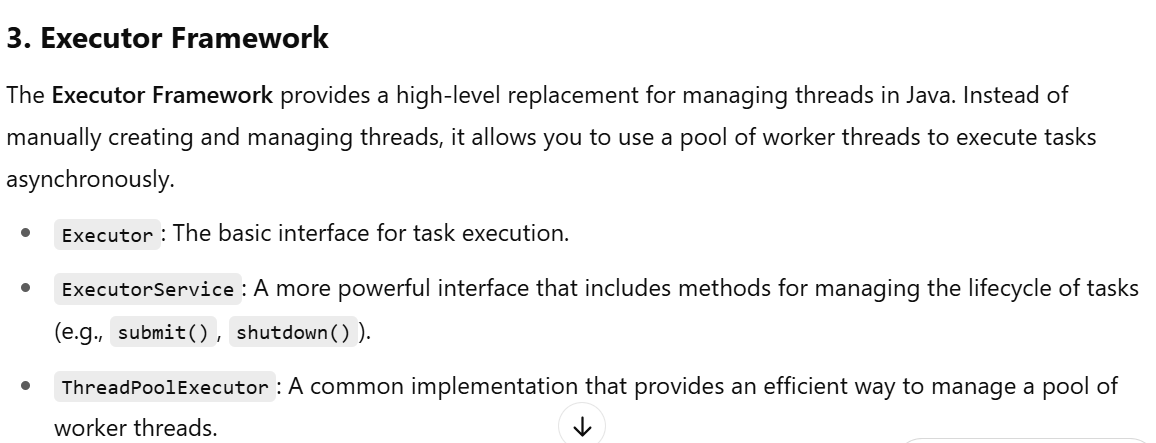
S- Single Responsibility Model(one class should have one responsibility)

O-open closed principle(class should be open for extension but not modification), polymorphism

L-Liscov Substitution principle(object of superclass should be replacable by obj of subclass without changing functionality)

I-Interface Segregation(write small interfaces instead of one big interface)

D-Dependency Inversion(high level module(business logic) should not rely on low level modules(databases) both should rely on interfaces or abstract classes)



A screen shot of a computer program

AI-generated content may be incorrect.

What is class loader in java?

In Java, a **ClassLoader** is a part of the Java Runtime Environment (JRE) responsible for **loading classes into memory** during runtime. It dynamically loads .class files as needed, rather than loading all classes at once during application startup.

**Types of ClassLoaders in Java**

Java uses a **hierarchical delegation model** for class loading. Here are the main types:

**1. Bootstrap ClassLoader**

* **Loads**: Core Java classes from rt.jar (like java.lang, java.util, etc.).
* **Written In**: Native code (not Java).
* **Parent of all class loaders**.

**2. Extension ClassLoader**

* **Loads**: Classes from the ext directory ($JAVA\_HOME/lib/ext).
* **Parent**: Bootstrap ClassLoader.

**3. System/Application ClassLoader**

* **Loads**: Classes from the application's classpath (CLASSPATH environment variable or -cp command-line option).
* **Parent**: Extension ClassLoader.

**4. Custom ClassLoaders**

* You can create your own by extending ClassLoader.
* Useful for:
  + Loading classes from encrypted files
  + Dynamic module loading
  + Plugin systems