Java

Java is a widely-used programming language for coding web applications. It has been a popular choice among developers for over two decades, with millions of Java applications in use today. Java is a multi-platform, object-oriented, and network-centric language that can be used as a platform in itself. It is a fast, secure, reliable programming language for coding everything from mobile apps and enterprise software to big data applications and server-side technologies.

Java is popular because it has been designed for ease of use. Some reasons developers continue to choose Java over other programming languages include:

I. High quality learning resources

Java has been around for a long time, so many learning resources are available for new programmers. Detailed documentation, comprehensive books, and courses support developers through the learning curve. In addition, beginners can start writing code in Core Java before moving to Advanced Java.

II. Inbuilt functions and libraries

When using Java, developers don't need to write every new function from scratch. Instead, Java provides a rich ecosystem of in-built functions and libraries to develop a range of applications.

III. Active community support

Java has many active users and a community that can support developers when they face coding challenges. The Java platform software is also maintained and updated regularly.

IV. High-quality development tools

Java offers various tools to support automated editing, debugging, testing, deployment, and change management. These tools make Java programming time and cost-efficient.

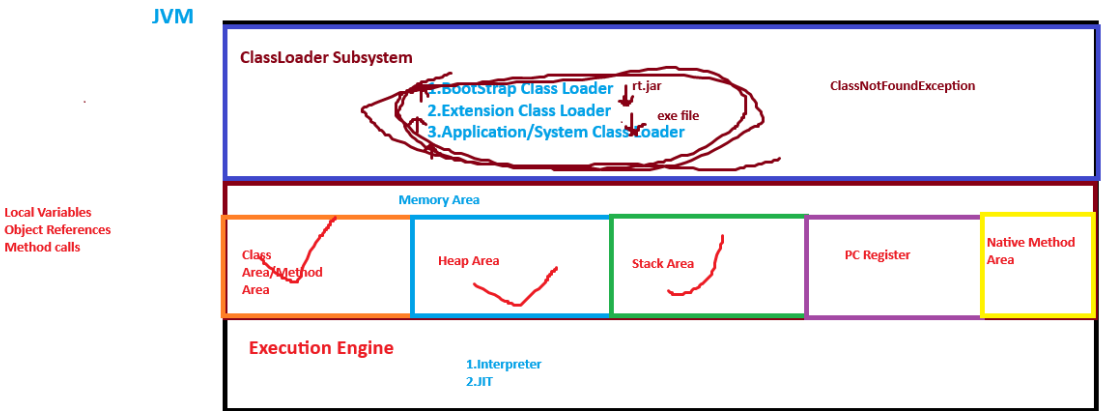
V. Platform Independent

Java code can run on any underlying platform like Windows, Linux, iOS, or Android without rewriting. This makes it especially powerful in today's environment, where we want to run applications on multiple devices.

VI. Security

Users can download untrusted Java code over a network and run it in a secure environment in which it cannot do any harm. Untrusted code cannot infect the host system with a virus nor can it read or write files from the hard drive. The security levels and restrictions in Java are also highly configurable.

The Java Virtual Machine (JVM)



The Java Virtual Machine (JVM) is a critical component of the Java Runtime Environment (JRE) that allows Java applications to run on any device or operating system without modification. The JVM provides a runtime environment in which Java bytecode is executed. Here are the key components and aspects of the JVM:

1. Class Loader Subsystem

The class loader subsystem is responsible for loading Java classes into memory. It performs three primary functions:

- Loading: Reads the .class files and brings the classes into memory.
- Linking: Performs verification, preparation, and resolution.
- Verification: Ensures the bytecode is valid and adheres to Java language specifications.
- Preparation: Allocates memory for class variables and initializes them to default values.
- Resolution: Converts symbolic references in the bytecode to actual references in memory.
- Initialization: Initializes class variables with the specified values and executes static blocks.

In Java, the class loader subsystem is responsible for loading classes into the Java Virtual Machine (JVM) dynamically at runtime. There are typically three types of class loaders in Java:

Bootstrap Class Loader:
This is the parent of all other class loaders in Java.
Responsible for loading core Java classes that are part of the Java runtime environment (JRE).
Implemented in native code and is not represented by a Java class.

Extension Class Loader (or Platform Class Loader):
Child of the Bootstrap Class Loader.

Loads classes from the JDK's extension directories (e.g., jre/lib/ext) as well as other system-wide library jars specified by the java.ext.dirs system property.

Application Class Loader (or System Class Loader):
Child of the Extension Class Loader.
Loads classes from the classpath specified by the java.class.path system property.
Responsible for loading application-specific classes and resources.
Implemented by the sun.misc.Launcher$AppClassLoader class.

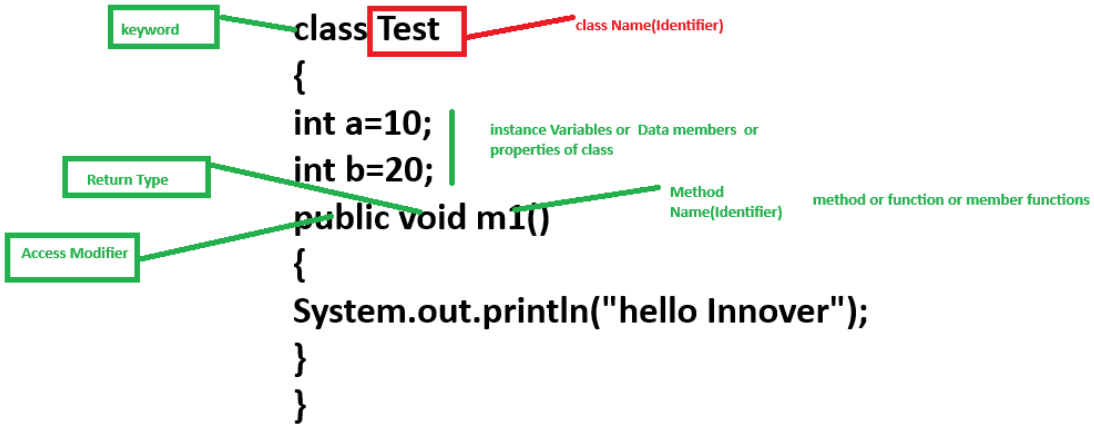
2. Runtime Data Areas

- The JVM organizes memory into several runtime data areas, each serving different purposes:
- Method Area: Stores class structures such as metadata, constant runtime pool, and method data.
- Heap: Stores objects and arrays. It's shared among all threads and is the runtime data area from which memory for all class instances and arrays is allocated.
- Stack: Each thread has a private JVM stack created at the same time as the thread. It stores frames, which hold local variables, operand stacks, and partial results.
- PC (Program Counter) Register: Each thread has its own PC register that stores the address of the current instruction being executed.
- Native Method Stack: Contains all the native methods used in the application. It is used when native methods

3. Execution Engine
The execution engine is responsible for executing the bytecode:

- Interpreter: Reads and executes bytecode instructions one at a time. While it is easy to implement and portable, it is slower because it interprets bytecode instructions every time.

Just-In-Time (JIT) Compiler: Improves performance by compiling bytecode into native machine code at runtime. The compiled code is cached and executed directly by the CPU, significantly increasing the execution speed.



Breaking down above code
Keywords:
class: Declares a new class (Test in this case).
public: Access modifier for the method m1, indicating that it can be accessed from anywhere.
void: Return type of the method m1, indicating that the method does not return any value.

Return type:
void: Indicates that the method m1 does not return any value.

Access modifier:
public: Specifies that the method m1 is accessible from any other class.

Class name:
Test: Name of the class being defined.

Instance variables:
int a = 10;: Defines an instance variable a of type int with an initial value of 10.
int b = 20;: Defines an instance variable b of type int with an initial value of 20.

Method name:
m1: Name of the method defined within the class Test.

In Java, there are several key concepts related to classes, methods, and variables that are essential to understand. Here's a detailed explanation of each term you mentioned:

1. **Keyword:**

Keywords in Java are reserved words that have predefined meanings and purposes within the language. They cannot be used as identifiers (such as variable names, method names, or class names) because they are already used by Java for specific operations or features. Here are some examples of keywords in Java:

Reserved for Classes and Interfaces: class, interface, enum
Reserved for Methods and Functions: void, return
Reserved for Data Types: int, boolean, double
Reserved for Flow Control: if, else, switch, case
Reserved for Loops: for, while, do, break, continue
Reserved for Exception Handling: try, catch, finally, throw, throws
Reserved for Access Modifiers and Others: public, private, protected, static, final, abstract, native, synchronized, volatile, transient, strictfp

2. Return Type:
In Java methods, the return type specifies what type of value the method will return when it finishes execution. If a method does not return any value, its return type is void. If it returns a value, the return type specifies the type of that value (e.g., int, double, String, etc.). Here are a few examples:

void: Indicates that the method does not return any value.
int: Indicates that the method returns an integer value.
double: Indicates that the method returns a double-precision floating-point number.
String: Indicates that the method returns a string.

3. Access Modifier:
Access modifiers in Java control the visibility or accessibility of classes, methods, and instance variables.

They define how classes and their members can be accessed or used by other classes or parts of the program. Java provides several access modifiers:
Public (public): Accessible from any other class.
Private (private): Accessible only within the same class.
Protected (protected): Accessible within the same package and by subclasses (even if they are in different packages).
Default (Package-private): If no access modifier is specified, it is accessible only within the same package.

4. Class Name:
In Java, a class is a blueprint or template for creating objects. The class name is the identifier used to define a particular class. Rules for naming a class:

- It must start with a letter (A-Z or a-z), currency character ($), or an underscore (_).

- Subsequent characters may be letters, currency characters, underscores, or digits (0-9).
- It cannot be a Java keyword or reserved word.

5. Instance Variable:
Instance variables (also called fields or member variables) are variables declared within a class but outside any method, constructor, or block. Each instance of the class (object) has its own copy of these variables. Instance variables can have different data types (like int, double, String, etc.) and can have default values if not explicitly initialized.

6. Method Name:
A method in Java is a collection of statements that perform some operation and are associated with a class or an object. Method names in Java follow the same rules as class names for naming conventions:

- They must start with a letter (A-Z or a-z), currency character ($), or an underscore (_).
- Subsequent characters may be letters, currency characters, underscores, or digits (0-9).
- They are case-sensitive and cannot be Java keywords or reserved words.

Java Inheritance
Inheritance is a fundamental concept in object-oriented programming that allows a new class (subclass) to be derived from an existing class (superclass). The subclass inherits all the fields and methods from the superclass, allowing code reuse and the creation of a hierarchical organization of classes.
In Java, a class can inherit from only one superclass, which is known as single inheritance. The extends keyword is used to inherit from a superclass.

```
class SuperClass {
   // fields and methods
}

class SubClass extends SuperClass {
   // inherits fields and methods from SuperClass
   // can also have its own fields and methods
}
```

Polymorphism

Polymorphism is the ability of an object to take on many forms. In Java, polymorphism is achieved through method overloading and method overriding.

Method Overloading

Method overloading is a feature that allows a class to have multiple methods with the same name but different parameters (number, types, or order of parameters). The compiler determines which method to call based on the arguments provided during method invocation.

```
OverloadingExample {
   void add(int a, int b) {
      System.out.println(a + b);
   }

   void add(int a, int b, int c) {
      System.out.println(a + b + c);
   }
}
```

Method Overriding

Method overriding is a feature where a subclass provides a specific implementation of a method that is already defined in its superclass. When the overridden method is called on an object of the subclass, the subclass's implementation is executed instead of the superclass's implementation.

```
class SuperClass {
   void show() {
      System.out.println("SuperClass show method");
   }
}

class SubClass extends SuperClass {
   @Override
   void show() {
      System.out.println("SubClass show method");
   }
}
```

Java Abstraction

Abstraction is a process of hiding the implementation details and showing only the essential features of an object or system. In Java, abstraction is achieved through abstract classes and interfaces.

Abstract Classes

An abstract class is a class that cannot be instantiated (cannot create objects of that class). It is used as a blueprint for other classes that inherit from it. Abstract classes can have abstract methods (methods without a body) and concrete methods (methods with a body).

```
class AbstractClass {
   // abstract method
   abstract void abstractMethod();

   // concrete method
   void concreteMethod() {
      System.out.println("Concrete method");
   }
}
```

Interfaces

An interface is a contract that defines a set of methods that a class must implement. Interfaces cannot have method bodies, but they can have default and static methods (with bodies) Interfaces are used to achieve abstraction and multiple inheritance in Java.

```
interface MyInterface {
   void method1();
   void method2();
}

class MyClass implements MyInterface {
   @Override
   public void method1() {
      // implementation
   }

   @Override
   public void method2() {
      // implementation
   }
}
```

Java Encapsulation

Encapsulation is the bundling of data and methods within a single unit (class). It is used to hide the internal implementation details and provide a well-defined interface to interact with the object.

In Java, encapsulation is achieved through the use of access modifiers (private, protected, and public) and getter and setter methods.

```
class EncapsulationExample {
   private int data; // private field

   public int getData() { // getter method
      return data;
   }
```

```
    public void setData(int value) { // setter method
        data = value;
    }
}
```

Rules for Java Method Overriding

When overriding a method in a subclass, there are certain rules to follow:

I. The method in the subclass must have the same name, return type, and parameter list as the method in the superclass.
II. The access modifier of the overriding method in the subclass cannot be more restrictive than the overridden method in the superclass.
III. The overriding method in the subclass can throw any unchecked exceptions or a subset of the checked exceptions thrown by the overridden method in the superclass.
IV. The overriding method in the subclass cannot be declared as static if the overridden method in the superclass is not static.

Constructors in Java

A constructor is a special method in a class that is used to initialize objects of that class. It has the same name as the class and does not have a return type (not even void).

Types of Java Constructors

1. Default Constructor: If no constructor is defined in a class, the Java compiler automatically creates a default no-argument constructor.
2. Parameterized Constructor: A constructor that accepts one or more parameters to initialize the object's state.
3. Copy Constructor: A constructor that creates a new object by copying the state of an existing object of the same class.

Constructor Overloading

Similar to method overloading, Java allows constructor overloading, where a class can have multiple constructors with different parameter lists. The compiler determines which constructor to call based on the arguments provided during object creation.

```
class ConstructorExample {
    int x, y;

    // Default constructor
    ConstructorExample() {
        x = 0;
        y = 0;
    }

    // Parameterized constructor
```

```
    ConstructorExample(int a, int b) {
        x = a;
        y = b;
    }
}
```

Interfaces in Java

An interface in Java is a blueprint or a contract that defines a set of methods and constants that a class must implement. Interfaces are used to achieve abstraction and multiple inheritance in Java.

```
interface MyInterface {
    int CONSTANT = 10; // constant
    void method1(); // abstract method
    void method2(); // abstract method

    // static and default methods (introduced in Java 8)
    static void staticMethod() {
        System.out.println("Static method");
    }

    default void defaultMethod() {
        System.out.println("Default method");
    }
}
```

A class that implements an interface must provide an implementation for all the abstract methods defined in the interface. Multiple interfaces can be implemented by a single class.

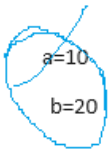
```
class MyClass implements MyInterface {
    @Override
    public void method1() {
        // implementation
    }

    @Override
    public void method2() {
        // implementation
    }
}
```

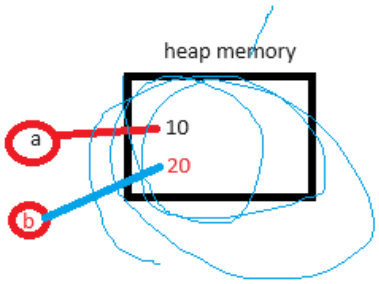
Interfaces are often used to define common behavior for different classes and can be helpful in achieving loose coupling between classes.

This covers the detailed explanation of Java Inheritance, Polymorphism (Method Overloading and Overriding), Abstraction, Encapsulation, Rules for Method Overriding, Constructors,

Constructor Overloading, and Interfaces. If you need any further clarification or have additional questions, feel free to ask.

```
class Test
{
int a;
int b;
public Test(int a, int b)
{
sop(this.a=a);
sop(this.b=b);
}
public int  m1(int a,int b)
{
return a+b;
}
}
class Main
{
public static void main(String args[])
{
```





object creation and the allocation of memory in the heap for objects in Java.

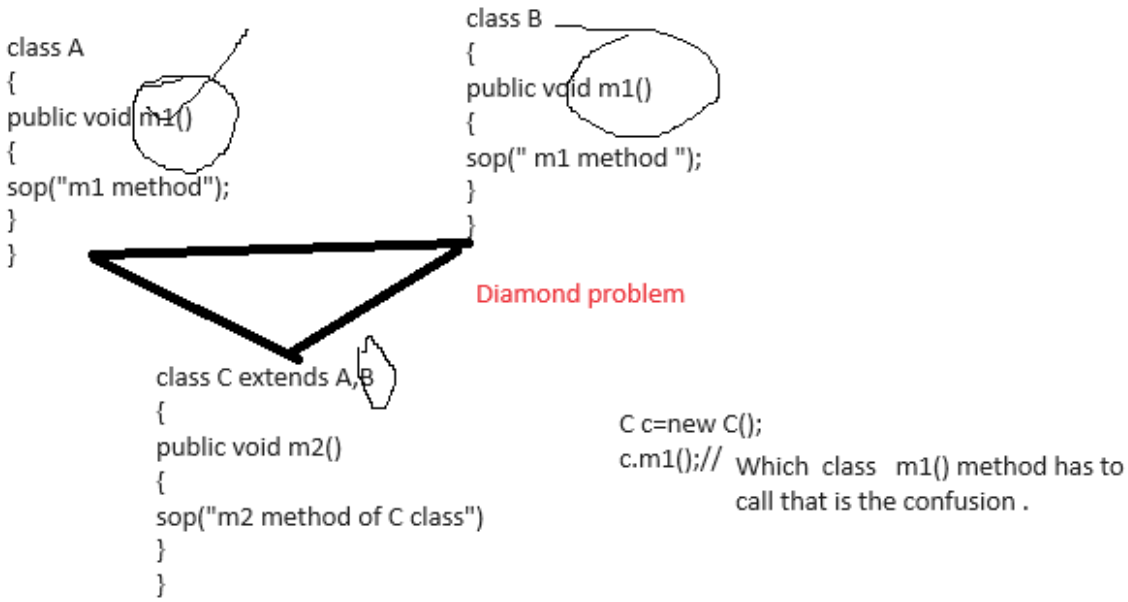
class Test:
This class has two instance variables: int a and int b.
It has a constructor Test(int a, int b) that initializes the instance variables using sop(this.a=a); and sop(this.b=b);.
It has a public method m1(int a, int b) that returns a+b.

class Main:
This class contains the main method, which is the entry point of the Java program.
Inside the main method, an instance of the Test class is created: Test t1 = new Test(10, 20);.

When the object t1 is created using the constructor Test(10, 20), the following happens:
Memory is allocated in the heap for the Test object.
The instance variables a and b are initialized with the values 10 and 20, respectively.
The values 10 and 20 are also stored in the heap memory.
The object reference t1 points to the memory location of the Test object in the heap.

The heap memory section in the image shows how the object t1 and its instance variables a and b are stored in the heap memory.

The image also shows the values 10 and 20 being passed as arguments to the constructor, and the values a=10 and b=20 being assigned to the instance variables using this.a=a and this.b=b inside the constructor.



the "Diamond Problem" in Java inheritance.
The Diamond Problem occurs when a class inherits from two classes that have the same superclass in the hierarchy. In this case, class C extends (inherits) from both class A and class B, which leads to the diamond problem.

class A:
This is the superclass.
It contains a public method m1().
The method sop("m1 method"); prints "m1 method" to the console.

class B:
This class also extends (inherits) from class A.
It has a public method m1().
Inside m1(), it prints "m1 method" using sop("m1 method");.

class C extends A, B:

Class C inherits from both class A and class B.
It has a public method m2().
Inside m2(), it calls sop("m2 method of C class") to print a string.
The confusion arises when an instance of class C is created (C c=new C();), and c.m1(); is called.
The problem is which class's m1() method should be called (class A's or class B's)

The Diamond Problem is a result of Java's single inheritance rule, which does not allow a class to inherit from multiple classes directly. This problem can be resolved by using interfaces or creating a new class that extends from one class and implements the other.

Exception Handling
Hierarchy of Exception classes:
In Java, exceptions are represented as objects derived from the Throwable class. There are two main types of exceptions:

1. Checked Exceptions: These are checked at compile-time. They extend Exception class and include exceptions that can be recovered from, such as IOException.

2. Unchecked Exceptions (Runtime Exceptions): These are not checked at compile-time. They extend RuntimeException and include programming errors, such as NullPointerException.

Both checked and unchecked exceptions derive from the Throwable class. Here's a simplified hierarchy:

Throwable
Error: Irrecoverable conditions that should not be caught (e.g., OutOfMemoryError).
Exception: Conditions that a well-written application should anticipate and recover from.
IOException, SQLException: Examples of checked exceptions.
RuntimeException: Unchecked exceptions like NullPointerException, ArrayIndexOutOfBoundsException.

Types of Exception
1. Checked Exceptions: Checked at compile-time, subclasses of Exception.
2. Unchecked Exceptions: Not checked at compile-time, subclasses of RuntimeException.

try-catch-finally
try: Block where exceptions are likely to occur.
catch: Block handles the exception caught.
finally: Block always executes after try and catch blocks.

Example:

```
try {
   // Code that may throw an exception
} catch (ExceptionType1 e1) {
   // Handle ExceptionType1
} catch (ExceptionType2 e2) {
   // Handle ExceptionType2
} finally {
   // Code that always executes, regardless of exceptions
}
```

throw:
throw: Used to manually throw an exception.
Example:

```
throw new SomeException("Custom error message");
```

throws:
throws: Declares the exceptions a method can throw.

Example:

```
void method() throws IOException, SQLException {
   // Method code
}
```

Multiple catch block:
Multiple catch blocks handle different exceptions.

Example:

```
try {
   // Code that may throw exceptions
} catch (IOException e) {
   // Handle IOException
} catch (SQLException e) {
   // Handle SQLException
}
```

Exception Handling with Method Overriding:
Subclass methods can declare exceptions that superclass methods don't.
Overriding methods can only throw unchecked exceptions or subclasses of exceptions declared in the superclass.

Example:

```
class Parent {
   void method() throws IOException {
```

```
    // Method code
  }
}

class Child extends Parent {
  @Override
  void method() throws FileNotFoundException {
    // Method code
  }
}
```

Java Collection Framework
Hierarchy of Collection Framework:

- Collection Interface: Root interface of the collection hierarchy.

I. List: Ordered collection (e.g., ArrayList, LinkedList).
II. Set: Collection without duplicates (e.g., HashSet, TreeSet).
III. Queue: Ordered for processing (e.g., PriorityQueue).

- Map Interface: Key-value pair (e.g., HashMap, TreeMap).

Collection interface:

- Collection: Base interface for all collection types.
- Iterator: Interface to iterate over elements in a collection.

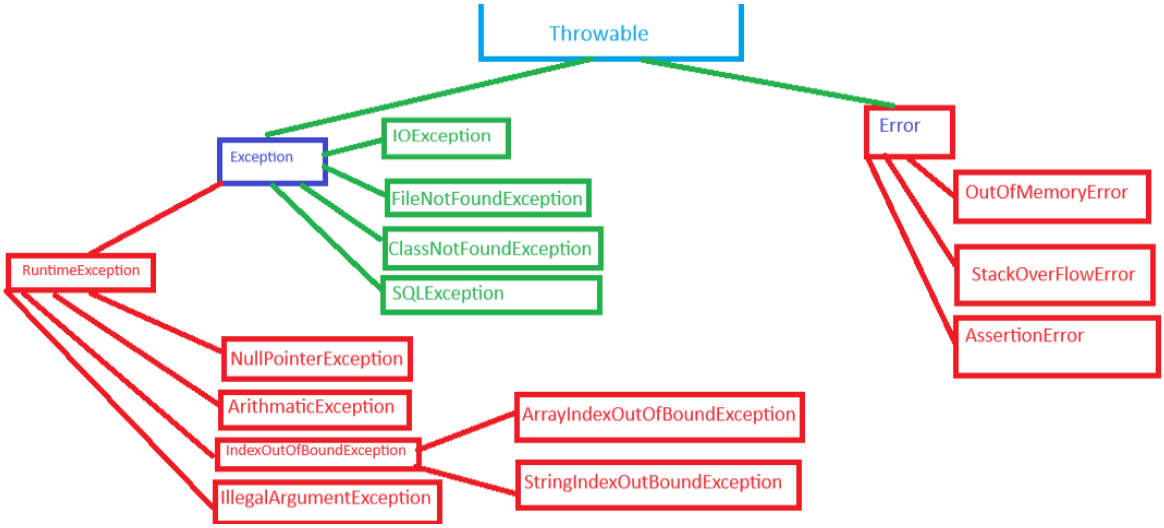
Set, List, Queue, Map, Comparator, Comparable interfaces:

- Set: Interface that does not allow duplicate elements (e.g., HashSet, TreeSet).
- List: Interface that maintains the order of elements (e.g., ArrayList, LinkedList).
- Queue: Interface to manage a collection for processing (e.g., PriorityQueue).
- Map: Interface to store key-value pairs (e.g., HashMap, TreeMap).
- Comparator: Interface to define custom sorting logic.
- Comparable: Interface to implement natural ordering of objects.

ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet:

- ArrayList: Dynamic array that can grow as needed.

- Vector: Similar to ArrayList but synchronized.
- LinkedList: Doubly-linked list for fast insertions and deletions.
- PriorityQueue: Queue where elements are ordered by priority.
- HashSet: Set that uses hashing for fast access.
- LinkedHashSet: Set that maintains insertion order.
- TreeSet: Set that maintains elements in sorted order.



hierarchy of exceptions

- **Throwable:**
  This is the root class of the exception hierarchy in Java. It is the superclass of all errors and exceptions.

  Exception: This is a subclass of Throwable and represents exceptions that can be caught and handled by the program.

I. IOException: This is a subclass of Exception and represents exceptions that occur during input/output operations, such as reading from or writing to files, network communication, etc.

II. FileNotFoundException: This is a subclass of IOException and is thrown when an attempt is made to open a file that does not exist.

III. ClassNotFoundException: This is a subclass of Exception and is thrown when an attempt is made to load a class that cannot be found.

IV. SQLException: This is a subclass of Exception and represents exceptions that occur during database operations using JDBC (Java Database Connectivity).

RuntimeException
This is a subclass of Exception and represents exceptions that occur during runtime, typically due to programming errors.

- NullPointerException: This is a subclass of RuntimeException and is thrown when an attempt is made to access a member (method or field) of a null object reference.
- ArithmeticException: This is a subclass of RuntimeException and is thrown when an exceptional arithmetic condition occurs, such as dividing by zero.
- IndexOutOfBoundsException: This is a subclass of RuntimeException and is thrown when an attempt is made to access an array or vector index that is out of the valid range.

I. ArrayIndexOutOfBoundsException: This is a subclass of IndexOutOfBoundsException and is thrown when an attempt is made to access an array index that is out of the valid range.

II. StringIndexOutOfBoundsException: This is a subclass of IndexOutOfBoundsException and is thrown when an attempt is made to access a String index that is out of the valid range.

- IllegalArgumentException: This is a subclass of RuntimeException and is thrown when an illegal or inappropriate argument is passed to a method.

Error: This is a subclass of Throwable and represents serious system-level errors that are typically not recoverable and cannot be handled by the program.

- OutOfMemoryError: This is a subclass of Error and is thrown when the Java Virtual Machine (JVM) runs out of memory and cannot allocate more memory for an object or array.
- StackOverflowError: This is a subclass of Error and is thrown when the JVM runs out of stack memory, typically due to an infinite recursion or a very deep nested loop.
- AssertionError: This is a subclass of Error and is thrown when an assertion (a boolean expression that is expected to be true) fails.

Lifecycle of a Thread

a thread goes through several states during its lifecycle. The main states are:

- New: When a thread object is created, it is in the new state.
- Runnable: After the start() method is called, the thread enters the runnable state and is eligible to be scheduled for execution by the operating system.
- Running: When the thread gets CPU time, it transitions to the running state, and its run() method is executed.
- Blocked/Waiting: A thread can enter the blocked or waiting state if it is waiting for a resource (e.g., lock, input/output operation) or a specific condition to occur.
- Terminated/Dead: After the run() method completes, the thread enters the terminated or dead state, and its lifecycle ends.

Thread Priority in Multithreading
Java allows you to set the priority of a thread using the setPriority() method. The priority determines the order in which threads are scheduled for execution by the operating system. Higher priority threads generally get more CPU time than lower priority threads. However, thread priorities are just hints to the operating system, and the actual scheduling is platform-dependent.

Runnable interface in Java
The Runnable interface is a functional interface that defines a single method, run(). It represents a task that can be executed by a thread. To create a new thread, you can either extend the Thread class and override the run() method, or implement the Runnable interface and pass an instance of it to the Thread constructor.

start() function in multithreading
The start() method is a non-static method of the Thread class. When you call start() on a thread object, it initiates the thread's lifecycle and moves it to the runnable state. The run() method is then executed when the thread gets CPU time.

Thread.sleep() Method in Java
The sleep() method is a static method of the Thread class. It causes the current thread to suspend execution for a specified period of time (in milliseconds). This method is often used to introduce delays or simulate real-time processes.

Thread.run() in Java
The run() method is the entry point for a thread's execution. It contains the code that will be executed when the thread starts running. You can override the run() method in a class that extends the Thread class or implement it in a class that implements the Runnable interface.

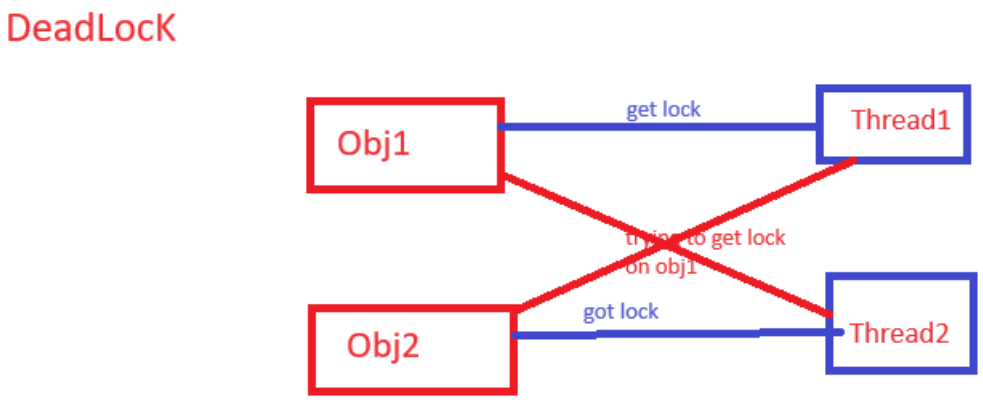
Deadlock in Java
A deadlock occurs when two or more threads are waiting indefinitely for resources held by each other, resulting in a standstill. This can happen when threads acquire multiple locks in different orders, leading to a circular wait condition.

Synchronization in Java
Synchronization is a mechanism in Java that allows threads to access shared resources safely, preventing data races and ensuring thread safety. Java provides different levels of synchronization

- Method-level lock When a method is declared as synchronized, only one thread can execute it at a time, acquiring an implicit lock on the object instance.
- Block-level lock: You can use the synchronized keyword to synchronize a block of code, allowing finer-grained control over thread synchronization.

Executor Framework in Java
The Executor Framework is a utility class in Java that provides a high-level abstraction for executing tasks asynchronously using thread pools. It simplifies the process of creating and managing threads, improving performance and scalability.

Callable Interface in Java
The Callable interface is similar to the Runnable interface but allows the task to return a value and throw a checked exception. It is used in conjunction with the Executor Framework to execute tasks that return results.



the concept of a "Deadlock" situation, which can occur in concurrent programming when two or more threads are waiting for each other to release resources that they need, resulting in a state of indefinite blocking. The diagram shows two objects (Obj1 and Obj2) and two threads (Thread1 and Thread2). Thread1 has acquired a lock on Obj1 and is trying to get a lock on Obj2, while Thread2 has acquired a lock on Obj2 and is trying to get a lock on Obj1. This circular waiting for resources leads to a Deadlock situation, where neither thread can proceed, and the program becomes stuck.