# JVM,JRE,JDK, Javac:

* Main method should be inside the public class.
* Public class name should me the name of the file.
* Only one class will be public in a java file.

1. **JVM (Java Virtual Machine)**:
   * The JVM is an abstract computing machine that enables Java bytecode to be executed on various platforms.
   * It provides the runtime environment in which Java bytecode can be executed.
   * The JVM is responsible for translating Java bytecode into machine-specific instructions and managing the execution of Java programs.
   * It provides features like memory management, garbage collection, and security.
2. **JRE (Java Runtime Environment)**:
   * The JRE is a package of software components that provides the runtime environment for Java applications.
   * It includes the JVM, libraries, and other components necessary for running Java applications but does not include development tools such as compilers and debuggers.
   * The JRE is required to run Java applications but does not include tools for developing or compiling Java code.
3. **JDK (Java Development Kit)**:
   * The JDK is a software development kit used for developing Java applications.
   * It includes the JRE, development tools such as the Java compiler (**javac**), debugging tools, libraries, and documentation.
   * Developers use the JDK to write, compile, debug, and run Java programs.
   * The JDK provides everything needed for Java development, including the tools and libraries required to create Java applications.
4. **javac**:
   * **javac** is the Java compiler, which is part of the JDK.
   * It is used to compile Java source code (.java files) into Java bytecode (.class files) that can be executed by the JVM.
   * **javac** translates human-readable Java source code into platform-independent bytecode.
   * Once compiled, the bytecode can be executed by any JVM, making Java programs cross-platform compatible.

# Naming Conventions in Java

* Class names should be **nouns**, in mixed cases with the **first** letter of each internal word capitalized. Interfaces names should also be capitalized just like class names.
* Methods should be **verbs**, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalized.
* Variable name can start with $,\_, and letters.
* Variable name is case sensitive.
* Variables, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalized.
* For constant, variable name should be defined in capital case letters.

# Memory Management:

There are two types of memory which JVM create in RAM. The memories are called stack, heap.

## Stack Memory:

The stack memory refers to a region of memory used for storing method frames. Each time a method is invoked, a new frame is pushed onto the stack, containing parameters, local variables, return address, and other necessary data for the execution of that method. When the method completes execution, its frame is popped off the stack.

* Stack memory store temporary variables and separate memory block for methods.
* It stores primitive data types.
* Store reference of objects created in heap.
* Each thread has its own stack memory.
* Variables within a scope is only visible and as soon as any variable goes out of scope, it gets deleted from the stack in LIFO order.
* When stack memory is full, it throws “java.lang.stackoverflowerror”.

## Heap memory:

* Heap memory store objects. There is no order of allocation the memory.
* Garbage collector is used to delete the unreferred objects from the heap.it uses the mark and sweep algorithm to delete the unreferred objects.
* Heap memory is store with all the threads. Heap also contain the string constant pool.
* Heap is divided into young and old generations. Minor GC runs in young. Major GC runs in the old generation.

## Garbage collector:

The garbage collector (GC) is a built-in mechanism responsible for automatically managing the memory used by objects created by a Java program. Its main purpose is to reclaim memory occupied by objects that are no longer referenced or needed, thus preventing memory leaks, and ensuring efficient memory utilization. This is managed by the JVM.

# Variables in Java

Now let us discuss different types of variables  which are listed asfollows:

* Local Variables(available only in a method)
* Instance Variables/ member variables.
* Static Variables/ class variable.

## Constant variables:

In Java, a constant variable is declared using the final keyword. Once initialized, the value of a constant variable cannot be changed during the program's execution. Constant variables are typically used to define values that should remain constant throughout the program.

## Primitive types:

1. Char :

The **char** data type in Java is used to represent a single 16-bit Unicode character. Unicode is a standard that assigns a unique numeric value to every character used in most writing systems used in the world today.

1 char is 16 bits.

Range- 0 to 65,535.

1. Byte:

The **byte** data type in Java is an 8-bit signed two's complement integer. It can hold values ranging from -128 to 127. **byte** is commonly used when dealing with raw binary data or when memory conservation is crucial.

1 byte is 8 bits.

Signed 2nd complement is used to represent the negative numbers.

Range- -128 to 127.

Default value is 0.

1. Short:

1 short is 2 byte means 16 bits.

Signed 2nd complement is used to represent the negative numbers.

Default value is 0.

1. int :

int is 4 bytes(32 bits)

Signed 2nd complement is used to represent the negative numbers.

Default value is 0.

1. Long :

Long is 8 bytes (64 bits).

Signed 2nd complement is used to represent the negative numbers.

Default value is 0.

1. Float:

The **float** data type is a single-precision 32-bit IEEE 754 floating-point. It can represent numbers with decimal points and is useful when dealing with values that don't require a high degree of precision. Floats are commonly used in applications where memory conservation is crucial or when dealing with large arrays of floating-point numbers.

1. Double:

The **double** data type is a double-precision 64-bit IEEE 754 floating-point. It provides greater precision compared to **float**, capable of representing a wider range of values with more significant digits. In Java, **double** is the default choice for representing floating-point numbers.

1. Boolean:

Boolean is 1 bit.

Default value is false.

## Types of conversions:

* Widening/automatic conversion:

Widening conversion occurs when a value of a smaller data type is automatically promoted to a larger data type.

It happens without any explicit casting because there's no loss of data.

For example, converting an int to a long or converting a float to a double.

* **Narrowing Conversion (Explicit Conversion or Casting)**:

Narrowing conversion occurs when a value of a larger data type is explicitly casted or converted to a smaller data type.

There's a potential loss of data, so explicit casting is required.

For example, converting a double to an int or converting a long to a short.

* Promotion during expression:

promotion during expression, also known as type promotion, occurs in Java when operands of different data types are used together in an expression. In such cases, the operands are automatically promoted to a common data type based on a set of rules defined by Java.

* Explicit casting during expression:

Explicit casting during expression, also known as type casting, is the process of converting a value from one data type to another data type by explicitly specifying the target data type. This is necessary when you want to convert a value from a larger data type to a smaller data type, which may result in a loss of precision or data.

## Reference types (non-primitive):

String:  
Strings in Java are considered reference types because they are implemented as objects. While they have some characteristics of primitive types (such as being immutable and having a literal representation), they are instances of the **String** class, which is part of the Java API.

In summary, strings are referred to as reference types in Java because they are implemented as objects, allocated on the heap, and accessed via references. While they have some characteristics of primitive types, such as being immutable and having literal representation, their behavior aligns more closely with that of reference types.

Interface:

Interfaces in Java serve as reference types, providing a blueprint for classes to adhere to without specifying implementation details. They define a contract of methods that implementing classes must fulfill, allowing for polymorphic behavior and dynamic binding at runtime. By declaring variables of interface types, Java enables flexibility in referencing objects of different implementing classes, promoting code extensibility and maintainability. With support for multiple inheritance of type, interfaces facilitate code organization and reuse, offering a powerful mechanism for abstraction and decoupling in Java programming.Top of Form

Bottom of Form

Top of Form

Bottom of Form

Array:

Arrays in Java are reference data types because they store references (or pointers) to the memory locations where the actual array elements are stored. When you declare an array variable, you're creating a reference to the array object rather than the array itself. This reference points to the memory location where the array is stored in the heap. Arrays in Java can hold elements of any data type, including primitive types and objects. Since arrays are objects themselves, they exhibit reference semantics, enabling dynamic binding, polymorphism, and flexibility in referencing arrays of different types and sizes. Therefore, arrays are considered reference data types in Java, providing a versatile mechanism for storing and manipulating collections of elements.Top of Form

# Operators:

Operator- this indicates what action to perform like addition, subtraction, etc.

Operand- this indicates the item, on which action must apply.

**Logical operators:**

It combines two or more conditions and returns true or false.

&& - Logical and.

|| - Logical or.

**Unary operators:**

Requires only single operand.

Increment (++)

Decrement(--)

Unary Minus (-) – Converts positive number to a negative number.

Unary plus(+) – Converts negative number to a positive number.

Logical not (!)

**Bitwise Operators:**

This operator works on bits(0 or 1). These operators are very fast.

& (and)

| (or)

^(XOR)

~(Not) -Output formula of not operator is –(n+1) when you apply not operator on the value n.

**Bitwise Shift Operator:**

It is used to shift the bits of a number to left or right.

<< (Signed Left Shift)

>>(Signed Right Shift)

>>>(Unsigned Right Shift)

There is no unsigned left shift- as it is same as signed left shift.

**Ternary operator:**

In Java, the ternary operator, also known as the conditional operator, provides a concise way to write conditional expressions. It takes three operands: a condition, a value to be returned if the condition evaluates to true, and a value to be returned if the condition evaluates too false. The syntax of the ternary operator is as follows:

**condition ? valueIfTrue : valueIfFalse**

**Type comparison operator:**

This is used to type check, whether object is of certain class or not.

**Instance of** is the keyword to check whether object is of certain class or not. This operations returns True or False.

# Wrapper class:

Wrapper classes in Java are classes that encapsulate primitive data types within an object. Each primitive data type (such as int, double, char, etc.) has a corresponding wrapper class (Integer, Double, Character, etc.) in Java.

**Compatibility with APIs**:

Many APIs and libraries in Java are designed to work with objects rather than primitive types. Wrapper classes allow primitive types to be used in such APIs without needing to convert them to objects manually.

**Additional Utility Methods**:

Wrapper classes provide additional utility methods that are not available for primitive types. These methods include **compareTo()**, **equals()**, **toString()**, etc., which can be useful in certain situations.

**Collections and Generics**:

Collections in Java (such as ArrayList, LinkedList, etc.) can only store objects, not primitive types. Therefore, if you need to store primitive types in a collection, you must use wrapper classes.

Bottom of Form

# Constructors:

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

In Java, a Constructor is a block of codes like the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. Every time an object is created using the new() keyword, at least one constructor is called.

**Note:** It is not necessary to write a constructor for a class. It is because the java compiler creates a default constructor (constructor with no arguments) if your class doesn’t have any.

Constructor do not have any return type. Constructor cannot be static or final or abstract, synchronized. Child class doesn’t have access to constructor of parent to override it. Constructor can be overloaded.

**How Java Constructors are Different from Java Methods?**

* Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
* Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
* Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

A private constructor is used in restricting object creation.This is often used in design patterns such as the Singleton pattern or Factory pattern to control object creation.

## Constructor chaining:

Constructor chaining in Java refers to the process of one constructor calling another constructor in the same class or its superclass. This is particularly useful for reducing code duplication and ensuring that common initialization logic is centralized.

**Within the same class**:

A constructor can call another constructor within the same class using the this() keyword.

Constructor chaining with **this()** in Java allows a constructor within a class to call another constructor within the same class. This enables the reusability of initialization logic and helps avoid code duplication by consolidating common setup tasks. When using **this()**, the call to another constructor must be the first statement in the constructor body. It's commonly used when there are multiple constructors in a class with varying parameters, allowing constructors with fewer parameters to delegate initialization tasks to constructors with more parameters. This technique promotes cleaner code organization and maintenance.

**From superclass constructors**:

A constructor in a subclass can call a constructor from its superclass using the super() keyword. constructor chaining with **super()** in Java allows a subclass constructor to call a constructor from its superclass. This is particularly useful when the superclass requires certain initialization logic to be performed before the subclass's specific initialization. By invoking **super()** with appropriate arguments, the subclass constructor can ensure that the superclass's initialization is executed before its own initialization. This mechanism supports the inheritance hierarchy and ensures proper initialization of objects across class hierarchies, facilitating modular and extensible code design.

# Access Modifiers:

1. **Default (Package-Private):**
   * No keyword is used.
   * Accessible only within the same package.
2. **Public:**
   * Accessible from any other class.
3. **Protected:**
   * Accessible within the same package and by subclasses, even if they are in different packages. Subclass means the child class.
4. **Private:**
   * Accessible only within the same class.

# Non-Access Modifiers:

1. **Final:**
   * Indicates that a class cannot be subclassed, or a method cannot be overridden.
2. **Abstract:**
   * Used in class and method declarations. An abstract class cannot be instantiated, and an abstract method must be implemented by any concrete (non-abstract) subclass.
3. **Static:**
   * Indicates that a method or variable belongs to the class rather than an instance of the class. It can be called without creating an instance of the class.
4. **Transient:**
   * Used to indicate that a variable should not be serialized when the class instance is serialized.
5. **Synchronized:**
   * Used to control access to a method or block of code by multiple threads. Ensures that only one thread executes the synchronized code at a time.
6. **Volatile:**
   * Indicates that a variable's value may be changed by multiple threads simultaneously. It ensures that a thread reads the most recent write to the variable.
7. **Native:**
   * Indicates that a method is implemented in platform-dependent code, typically written in another programming language like C or C++.

# Strings:

Strings are the types of objects which can store characters as elements.

## Memory Allotment of String

Whenever a String Object is created as a literal, the object will be created in the String constant pool. This allows JVM to optimize the initialization of String literal.

The string can also be declared using a **new** operator i.e. dynamically allocated. In case of String that are dynamically allocated they are assigned a new memory location in the heap. This string will not be added to the String constant pool.

If you want to store this string in the constant pool, then you will need to “intern” it. It is preferred to use String literals as it allows JVM to optimize memory allocation.

## String

String is an immutable class which means a constant and cannot be changed once created and if wish to change , we need to create a new object and even the functionality it provides like toupper, tolower, etc. all these return a new object , it will not modify the original object. It is automatically thread safe.

## String Buffer

String buffer  is a peer class of **String**that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while string Buffer represents growable and writable character sequences means it is mutable in nature and it is thread safe class , we can use it when we have multi-threaded environment and shared object of string buffer i.e., used by multiple thread. As it is thread safe so there is extra overhead, so it is mainly used for multithreaded program.

## StringBuilder

String builder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters and it is not thread safe and its used only within the thread , so there is no extra overhead , so it is mainly used for single threaded program. String Buffer and StringBuilder are similar, but StringBuilder is faster and preferred over String Buffer for the single-threaded program.

# Object-oriented programming (OOPS):

The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

## Why Java is not a purely Object-Oriented Language?

Java is not considered a purely object-oriented language due to its support for primitive data types, static members, and the existence of wrapper classes. Here's how Java deviates from the principles of a pure object-oriented language.

1. All predefined types are objects**:** In Java, primitive data types such as int, long, Boolean, float, char, etc., are not treated as objects. This means that while Java provides classes like Integer and Float as wrapper classes to allow primitive types to be treated as objects, the language inherently distinguishes between primitive types and objects.
2. All operations performed on objects **must be only through methods exposed at the objects:** Java allows certain operations to be performed directly on primitive types, such as arithmetic operations, without requiring them to be encapsulated within methods of objects. Additionally, the presence of static methods and variables in Java allows for operations to be performed without the need for object instantiation.

While Java incorporates many object-oriented features, its inclusion of primitive types and support for non-method-based operations prevent it from meeting the strict criteria for purity in object-oriented design.

## Encapsulation(informationhiding)

Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

## Abstraction(implementation hiding)

Abstraction refers to the concept of hiding complex implementation details and showing only the necessary features of an object to the outside world. It allows you to focus on what an object does rather than how it does it. In Java, abstraction is achieved through abstract classes and interfaces.

* In Java, abstraction is achieved by[**interfaces**](https://www.geeksforgeeks.org/interfaces-in-java/)and [**abstract classes**](https://www.geeksforgeeks.org/abstract-classes-in-java/). We can achieve 100% abstraction using interfaces.
* There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/).
* An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

### Abstract Class:

An abstract class is a class that cannot be instantiated on its own and may contain one or more abstract methods. Abstract methods are methods without a body, which are meant to be implemented by the subclasses. Abstract classes can also contain concrete methods, i.e., methods with an implementation.

### Interfaces:

Interfaces are another method of implementing abstraction in Java. The key difference is that, by using interfaces, we can achieve 100% abstraction in Java classes. In Java or any other language, interfaces include both methods and variables but lacks a method body.

## Polymorphism

Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, so it means many forms.

### Compile-time Polymorphism(Method Overloading)

When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments but not on the return type.

### Runtime Polymorphism (Method overriding)

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the subclass has the same name, return type, and parameter list as the method in the superclass. When a method is called on an object, the Java runtime system determines which version of the method to execute based on the actual type of the object at runtime.

The [access modifier](https://www.geeksforgeeks.org/access-modifiers-java/) for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass. Doing so will generate a compile-time error.

### Advantages of Polymorphism in Java

1. Increases code reusability by allowing objects of different classes to be treated as objects of a common class.
2. Improves readability and maintainability of code by reducing the amount of code that needs to be written and maintained.
3. Supports dynamic binding, enabling the correct method to be called at runtime, based on the actual class of the object.
4. Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.

### Disadvantages of Polymorphism in Java

1. Can make it more difficult to understand the behavior of an object, especially if the code is complex.
2. This may lead to performance issues, as polymorphic behavior may require additional computations at runtime.

## Inheritance

Inheritance is the mechanism in Java by which one class is allowed to inherit the features(attributes and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and variables of that class. In addition, you can add new fields and methods to your current class as well.

**Note:** In Java, constructor of the base class with no argument gets automatically called in the derived class constructor.

But, if we want to call a parameterized constructor of the base class, then we can call it using super(). The point to note is base class constructor call must be the first line in the derived class constructor.

### Why Inheritance?

* **Code Reusability:**The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
* **Method Overriding:**[Method Overriding](https://www.geeksforgeeks.org/overriding-in-java/) is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
* **Abstraction:**The concept of abstract where we do not have to provide all details is achieved through inheritance. [Abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2/)only shows the functionality to the user.

### Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance

In single inheritance, subclasses inherit the features of one superclass.

1. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.

1. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.

1. Multiple Inheritance

In [Multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/), one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support [multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. In Java, we can achieve multiple inheritances only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/).

1. Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn’t support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) if we want to involve multiple inheritance to implement Hybrid inheritance. However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.

## Objects relationships:

### Is-a" Relationship (Inheritance):

In object-oriented programming, an "is-a" relationship, also known as inheritance, represents a relationship where a subclass (derived class) shares the same behavior and attributes as its superclass (base class). The subclass is a specialized version of the superclass. This relationship is typically expressed using the extends keyword in Java.

example:

Circle and Rectangle are subclasses of Shape.

Both Circle and Rectangle inherit common attributes and methods from the Shape superclass.

We can say "a Circle is a Shape" and "a Rectangle is a Shape".

### "Has-a" Relationship:

A “has-a” relationship represents a relationship where one class contains an instance of another class as one of its members. This implies that the containing class has a component or part of another class. It's more of a "has-a" or "contains-a" relationship.

example:

* The **Car** class has a private member variable **engine**, representing an instance of the **Engine** class.
* The **Car** class creates an instance of **Engine** in its constructor.
* We can say "a Car has an Engine".

#### **Aggregation (weak relation)**

Aggregation is a type of "has-a" relationship where one class contains another class, but the contained class does not solely belong to the containing class. It means that the contained class can exist independently of the containing class. Aggregation is often represented by a one-to-many relationship.

**Example**: Consider a university having multiple departments. Each department consists of multiple students. Here, the university "has" departments, and each department "has" students. However, students can exist independently of their department.

#### **Composition (Strong relation)**

Composition is a stronger form of aggregation where the lifetime of the contained object is controlled by the containing object. If the containing object is destroyed, all its contained objects are also destroyed. Composition is often represented by a one-to-one relationship.

**Example**: Consider a house containing rooms. The existence of rooms is entirely dependent on the existence of the house. If the house is demolished, all its rooms are destroyed as well.

# Methods:

Method is used to perform certain task. It’s a collection of instruction that perform some specific tasks. It can be used to bring code the readability and reusability.

## Static methods:

These methods are associated with the classes.

Can be called just with the class name.

static methods cannot be overridden. When you attempt to define a static method with the same signature in a subclass, it's not considered overriding, but rather method hiding. This is because the subclass's static method with the same name and signature simply hides the superclass's static method, but it's not polymorphic in the same way instance methods are.

static methods can be overloaded in Java.

Static methods cannot access Non static methods and instance variables.

When to declare a method static?

Method which does not modify the state of the object can be declared static.

Utility method which does not use any instance variable and compute only on arguments.

## Final method:

Final method can’t be overridden. Final method means its implementation cannot be changed. If child class cannot change its implementation, then no use of overridden.

## Abstract method:

it is only defined in abstract class. Only method declaration is done in abstract class and Its implementation is defined in child classes.

## Variable arguments:

* Variable number of inputs in the parameter.
* Only one variable argument can be present in the method.
* It should be the last argument in the list of arguments.

# Class:

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties(attributes) and behavior(methods) that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code.

## Concrete class:

A concrete class in object-oriented programming is a class that provides a complete implementation of all its methods and can be instantiated to create objects. Unlike abstract classes, concrete classes can be directly instantiated to create objects because they provide concrete implementations for all their methods. Examples of concrete classes include **String**, **ArrayList**, **HashMap**, and user-defined classes with complete method implementations.

## Abstract class:

An abstract class in object-oriented programming serves as a blueprint for other classes to inherit from. It contains one or more abstract methods, which are declared but not implemented in the abstract class itself. Abstract methods are meant to be implemented by subclasses. Abstract classes cannot be instantiated on their own; they serve as templates for other classes to extend and provide specific implementations.

* Abstract classes have both abstract and non-abstract methods.
* We cannot create instance of this class.
* Constructors can be created inside them. And with super keyword from child class, we can access them.

## Super and sub-class:

A class that is derived from another class is called a subclass. The class from where the subclass is derived is called a super class. In java, in the absence of any other explicit super class, every class is implicitly a subclass of OBJECT class. Object is the topmost class in java. Object has some common methods like clone(), toString(),equals(),notify(),wait(),etc

## Nested class:

A class within another class is called nested class.

### Static Nested class:

In Java, a static nested class, also known as a static inner class, is a class that is defined within another class but marked with the **static** keyword. Unlike non-static (or member) inner classes, static nested classes do not have access to the instance variables and methods of the enclosing class. They are essentially independent of any specific instance of the outer class and can be instantiated without an instance of the outer class.

### Inner class (Non-Static Nested class):

An inner class, also known as a non-static nested class, is a class declared within another class in Java. Unlike static nested classes, inner classes have access to the instance variables and methods of the outer class, even if they are private. Inner classes are commonly used to logically group classes that are only used in one place, increase encapsulation, and improve code organization.

#### **Anonymous inner class:**

An anonymous inner class in Java is a type of inner class that doesn't have a name and is declared and instantiated at the same time. It's typically used when you need to create a one-time-use class with a small piece of functionality, often for implementing interfaces or extending classes without explicitly creating a new named class.

#### **Member Inner class:**

In Java, a member inner class, also known as a non-static nested class, is a class defined within another class. Unlike static nested classes, member inner classes have access to the instance variables and methods of the enclosing class. They are associated with an instance of the outer class and cannot be instantiated without an instance of the outer class.

#### **Local inner class:**

In Java, a local inner class is a class defined within a method or a code block (such as a loop or an if statement) of another class. Unlike member inner classes, local inner classes are scoped to the block in which they are defined and cannot be accessed outside of that block. They have access to local variables, parameters, and constants of the enclosing method or block, but they cannot access non-final local variables unless they are effectively final (meaning their value doesn't change after initialization).

## Generic class:

A generic class is a class in object-oriented programming that is parameterized with one or more type parameters. These type parameters allow you to create classes and methods that can work with any data type, rather than being restricted to a specific type.

Here's a breakdown of the key concepts related to generic classes:

Type Parameters: Type parameters are placeholders for types that are defined when the class is instantiated or when a method is called. These parameters are specified within angle brackets < > following the class name or method signature.

Flexibility: Generic classes provide flexibility by allowing you to define classes and methods that can work with a variety of data types without sacrificing type safety. This means you can write code that is more reusable and versatile.

Type Safety: Although generic classes provide flexibility, they still maintain type safety. The compiler ensures that you only use the types specified when instantiating the generic class or calling its methods. This helps catch type-related errors at compile time rather than at runtime.

### Inheritance with a Non-Generic Subclass:

* When a non-generic subclass inherits from a generic superclass, the subclass typically operates on a specific type, ignoring the generic nature of the superclass.
* The subclass can use the generic functionality provided by the superclass, but it doesn't need to specify type parameters since it operates on a fixed data type.
* The subclass can introduce its own methods or properties without being constrained by the generic type parameters of the superclass.

### Inheritance with a Generic Subclass:

When a generic subclass inherits from a generic superclass, it can either reuse the type parameters of the superclass or introduce its own.

The generic subclass can add additional type parameters or constraints, providing further flexibility or specialization.

It can override methods of the superclass and implement its own generic methods or properties.

### Bounded generics

Bounded generics in Java allow you to restrict the types that can be used as type arguments in generic classes or methods. This restriction is based on certain criteria, such as requiring the type of argument to extend a specific class or implement a particular interface. Bounded generics help enforce stronger type safety and enable you to utilize the functionalities provided by the bounded types. There are two main types of bounds in Java generics: upper bounds and lower bounds.

#### **Upper Bounded Generics:**

Upper bounded generics restrict the type arguments to be a subtype of a specified class or to implement a specific interface.

#### **Multi-bounded generics**

Multi-bounded generics in Java allow you to specify more than one bound for a type of parameter. This means you can enforce that the type of argument must extend multiple classes or implement multiple interfaces simultaneously. Multi-bounded generics are useful when you need type parameters to satisfy multiple criteria.

### Wildcard:

Wildcard types in Java generics provide a flexible way to work with unknown types or types that are related in a certain way without specifying the exact type. Wildcards are denoted by the ? symbol.

Wildcard types are particularly useful in method parameters, where you want to accept arguments of a certain type or its subtypes without specifying the exact type. They enhance the flexibility and reusability of generic code by allowing it to work with a wider range of types.

#### **unbounded wildcard**

An unbounded wildcard represents an unknown type. It can be used when you don't care about the actual type of the elements.

#### **upper bounded wildcard**

An upper bounded wildcard represents any type that is a subtype of a specified type (**Type**) or implements a specified interface.

It allows you to work with a collection of objects of a specific type or its subtypes.

#### **lower bounded wildcard**

A lower bounded wildcard represents any type that is a supertype of a specified type (**Type**) or is the specified type itself.

It allows you to work with a collection of objects of a specific type or its supertypes.

## POJO class:

* Pojo stands for plain old java object.
* They contain variables and its getter and setter methods.
* Class should be public.
* Public default constructor.
* No annotations should be used like @table,@Id,etc.
* It should not extend any class or implement any interfaces.

## ENUM class:

In Java, Enums (enumerations) are special data types used to define a set of predefined constants. Each constant in an Enum is an instance of the Enum type and has its own identity. Enums are typically used to represent a fixed set of related constants, such as days of the week, months of the year, or error codes.

* It has a collection of constants(variables which values cannot be changed)
* Its constants are static and final implicitly.
* It cannot extend any class, as it internally extends java.lang.Enum class.
* It can implement interfaces.
* It can have variables, constructors, methods.
* It cannot be instantiated (as its constructor will be private only, even you give default, in bytecode it will make private).
* No other class can extend Enum class.
* It can have abstract method, and all the constants should implement that abstract method.
* Common functions used on the constants. Values(), Ordinal(), Valueof(), name().
* Method overriding in Enum classes allows you to provide specific implementations of methods for each constant (instance) of the Enum. This can be useful when you want each constant to have its own behavior or properties.
* In Java, interfaces can be implemented by enums just like they can be implemented by classes. This allows enums to provide specific behavior for each constant (enum instance) by implementing methods defined in the interface.
* enums are versatile constructs that offer numerous benefits, including type safety, readability, compile-time checking, and support for various design patterns and use cases. They are an essential part of Java programming and are widely used in various applications and libraries.

## Final class:

A final class is a class that cannot be subclassed or extended by other classes. When a class is declared as final, it means that its design is complete, and it cannot be further modified or extended. This provides a level of control over the class's behavior and ensures that its functionality remains consistent.

## Singleton class:

This class main objective is to create only one object. The purpose of a singleton class is to ensure that a class has only one instance and provides a global point of access to that instance.

### Eager initialization:

Eager initialization of a singleton refers to creating the instance of the singleton class at the time of class loading or when it is first referenced, rather than lazily creating it when it is needed. This ensures that the singleton instance is available immediately and is always ready for use. In this approach, as soon as we start the application the singleton object will be created and loaded into memory.

### Lazy initialization:

Lazy initialization of a singleton refers to creating the instance of the singleton class only when it is first requested, rather than eagerly creating it at the time of class loading or when it is first referenced. This can be beneficial in scenarios where the singleton instance might not always be needed, and lazy initialization can help save resources by deferring the creation until necessary.

**Concurrency Issues**: Lazy initialization can introduce concurrency issues in multi-threaded environments. If multiple threads attempt to access the singleton instance simultaneously before it has been initialized, there's a possibility of race conditions or inconsistent behavior. Synchronization mechanisms such as locks or mutexes may be required to ensure thread safety, which can further impact performance.

### Synchronized initialization:

Synchronized initialization of a singleton is a technique used to ensure thread safety when initializing the singleton instance, particularly in multi-threaded environments. This approach synchronizes the creation of the singleton instance to prevent multiple threads from simultaneously creating separate instances. By synchronizing the initialization process, synchronized initialization ensures that only one singleton instance is created, even in multi-threaded environments, thereby avoiding race conditions and ensuring thread safety. However, it's essential to be mindful of potential performance implications, as synchronization introduces overhead due to locking mechanisms. This method is very slow as it locks and unlocks the method for every single thread, and this results in slow performance.

### Double locking:

In the context of singleton pattern implementation, double locking is used to ensure that only one instance of the singleton is created while minimizing the overhead of synchronization. It typically involves checking if the singleton instance exists and, if not, acquiring a lock to prevent concurrent creation of multiple instances.

Double locking helps reduce the overhead of synchronization by minimizing the time spent holding the lock. However, it's important to note that the use of double locking should be approached with caution, as it can be tricky to implement correctly and may not be necessary in all situations.

### Double locking with volatile:

double-checked locking with **volatile** is a technique used to implement lazy initialization of a singleton while ensuring thread safety and minimizing synchronization overhead. The **volatile** keyword ensures that changes made to the shared variable are visible to all threads immediately.

Using double-checked locking with **volatile** ensures that lazy initialization of the singleton is thread-safe and minimizes the overhead of synchronization by only synchronizing the critical section where the instance is created. However, it's crucial to note that prior to Java 5, the **volatile** keyword didn't guarantee safe publication, and other mechanisms such as the **synchronized** keyword or using **enum** for singleton implementation were recommended. Since Java 5, **volatile** has been enhanced to provide safe publication semantics, making double-checked locking with **volatile** a viable option for lazy initialization of singletons in Java.

Top of Form

### Bill pugh singleton:Bottom of Form

Bill Pugh's solution, also known as the "Initialization-on-demand holder idiom," is a technique for implementing lazy initialization of singletons in Java. It leverages the static inner class to delay the instantiation of the singleton until it is accessed for the first time. This approach is thread-safe without requiring explicit synchronization.

Bill Pugh's solution is efficient and thread-safe because the Java language specification guarantees that a class will not be initialized until it is accessed. Therefore, the **SingletonHelper** class is only loaded and initialized when the **getInstance()** method is called for the first time. This approach avoids the overhead of synchronization and ensures that the singleton instance is created only when it is needed.

## Immutable class:

* we cannot change the value of the object once its created.
* Declare class as final so that it cannot be extended.
* All class members should be private. So that direct access can be avoided.
* And class members are only initialized only once using constructor.
* There should not be any setter methods, which are generally used to change the values.
* Just getter methods. And returns copy of member variable.
* Example: String, wrapper classes, etc.

# Interfaces:

* interface is a way to define a contract or a blueprint for classes to follow.
* Interfaces declare a set of methods that a class implementing the interface must provide. In simple terms we can say that it helps to achieve abstraction.
* Only public and default modifiers are allowed (protected and private are not allowed).
* Interface can be a data type. We cannot create object of an interface, but it can hold the reference of all the classes which implements it. And at runtime, it decides which method need to be invoked.
* In java multiple inheritance is only possible through interfaces and it’s not possible through classes.

## Methods in interfaces:

In interfaces all methods are implicit public only. Methods can’t be declared as final.

## Fields in interfaces:

Fields in interfaces are public, static, and final implicitly (constants).

You can make fields private or protected.

## Interface implementation:

* Overriding method cannot have more restrict access specifiers.
* Concrete class must override all the methods declared in the interfaces.
* A class can implement multiple interfaces.
* Abstract classes are not forced to override all the methods.

## Nested interfaces:

Nested interfaces can be declared within another interface or class. Generally, it is used to group, logical related interfaces.

**Rules:**

* A nested interface declared in another interface must be public.
* A nested interface within a class can have any access modifier.
* When you implement outer interface, inner interface implementation is not required and vice versa.

## Interface vs Abstract class: A black and white screen with white text Description automatically generated

## Java 8 interface features:

### Default method :

Before java 8, interfaces can only have abstract method. And all Childs class must provide abstract method implementation. One of the main advantages of default methods in interfaces is backward compatibility. Before Java 8, adding new methods to an interface could break existing implementations, as those implementations were forced to provide concrete implementations for all methods in the interface. This limitation made it challenging to evolve interfaces without breaking existing codebases. With default methods, interface designers can add new methods to interfaces without breaking the existing implementations.

**Limitations** when using default methods:

* **Name conflicts**: If a class inherits multiple interfaces with default methods having the same signature, a compilation error occurs. In such cases, the class must explicitly override the conflicting method.
* **Diamond problem**: If a class inherits multiple interfaces that have default methods with the same signature but different implementations, the class must explicitly resolve the conflict by overriding the method.Top of Form

### Static method (public static):

* We can provide the implementation of the method in the interface with the static method.
* But it cannot be overridden by the classes which implements the interface.
* We can only access it by using interface name itself where it was defined.
* Static methods in interfaces are by default public.

## Java 9 features:

### Private and private static method:

* We can provide the implementation of method but as a private access modifier in interface.
* It brings code readability. For example, if multiple default methods share some code, it can be created as a private or private static method.
* Private method can be static or non-static.
* From static method, we can only call private static interface method.
* Private static method can be called by both static and non-static methods.
* Private interface methods cannot be abstract. Means we must provide the definition.
* It can be used inside the interface only. Private and private static methods are available to use only in the interface.

## Functional interfaces:

If an interface contains only one abstract method, that is known as functional interface. It’s also known as single abstract interface. @FunctionalInterface keyword can be used at the top of the interface(optional). In functional interfaces we can have other methods like default, static, private methods, we can inherit objects from other class like from the object class.

Note: Functional interfaces can be implemented in 3 ways. By implementing via a class, using the anonymous class, using the lambda expressions.

### Lambda expressions:

Lambda expression is a way of implementing the abstract method of the functional interface.

Lambda expressions, often referred to simply as "lambdas," are a concise way to create anonymous functions in programming languages. These functions are anonymous because they don't have a name like regular functions do. Instead, they are defined inline and are usually used for short, simple operations. Lambda expression can only work on the abstract method of the function interface.

### Types of functional interfaces:

Presents in package – java.util.function.

Consumer:

Represents an operation that accept a single input parameter and returns no result.

Supplier:

Represent the supplier of the result. Accept no parameter but produce a result.

Function:

Represent a function, that accept one argument process it and produce a result.

Predicate:

Represent a function, that accept one argument and returns a Boolean.

# Reflection of classes:

In Java, reflection is a powerful feature that allows you to inspect and manipulate classes, interfaces, fields, methods, and constructors at runtime. Reflection enables you to access information about the structure of classes and their members, even if that information wasn't known at compile time.

To reflect the class, we first need to get an object of the class.

What is this class Class?

Instance of the class Class represents classes during runtime. JVM creates one Class object for every class which is loaded during the runtime. This Class object has meta data info about the particular class like its method, fields, constructor, etc.

How to get the particular class Class object?

There are 3 ways.

1. Using forName() method.
2. Using .class method.
3. Using getClass() method.

How to reflect methods in java and access its meta data ?

Reflection of methods gives us the meta data about the methods. There are methods that we can perform on the class objects that we can create using the above given 3 ways. This methods on class objects will help us in getting the meta data.

## How to invoke the methods using reflection?

In Java, you can invoke methods dynamically using reflection by obtaining a **Method** object representing the method you want to invoke and then calling the **invoke()** method on that **Method** object.

**Obtain the Class Object**: First, obtain a **Class** object representing the class containing the method you want to invoke. You can obtain this **Class** object using class literals or by calling the **getClass()** method on an instance of that class.

**Get the Method Object**: Next, obtain a **Method** object representing the method you want to invoke. You can obtain this **Method** object using methods such as **getMethod()** or **getDeclaredMethod()** on the **Class** object.

**Invoke the Method Dynamically**: Finally, you can invoke the method dynamically using the **invoke()** method on the **Method** object. Pass the instance on which you want to invoke the method as the first argument, followed by any method arguments.

## Change public and private field values:

In Java, reflection provides the ability to access and modify private and public fields of a class dynamically at runtime. To change the value of a field using reflection, you first obtain a **Field** object representing the field you want to modify by calling methods such as **getField()** or **getDeclaredField()** on the **Class** object. If the field is private, you need to set it accessible using the **setAccessible(true)** method on the **Field** object to bypass access restrictions. Then, you can use the **set()** method on the **Field** object to set the new value of the field. This method takes the object instance on which you want to modify the field value as the first argument, followed by the new value.

## Reflection of constructors:

In Java, reflection enables the dynamic examination and manipulation of constructors at runtime. Through reflection, you can obtain **Constructor** objects representing the constructors of a class, allowing you to inspect their metadata and instantiate objects of that class dynamically. This is particularly useful when you need to create instances of a class without knowing the constructor's signature at compile time or when dealing with classes whose constructors have restricted access, such as private constructors. By using reflection, you can bypass access restrictions, instantiate objects, and perform operations that would otherwise be challenging or impossible using conventional programming techniques. However, reflection should be used judiciously due to its potential impact on performance and code readability, and developers should be mindful of the security implications associated with dynamically accessing and manipulating constructors at runtime. Top of Form Bottom of Form

Using the reflection we can break thew singleton design pattern.

Why we try to avoid refection?

1. It breaks the oops principle of encapsulation itself.
2. Reflection is slow. It is slow because it happens at runtime.
3. It increases the complexity of understanding.

# Annotation:

* It is kind of adding the metadata to the java code.
* Means its usage is optional.
* We can use this metadata information at runtime and can add certain logic in our code if wanted.
* How to Read metadata information? Using reflection as we discussed earlier.
* Annotations can be applied at anywhere like classes, methods, interfaces, fields, parameters, etc.

## Annotation used on java code:

### @Depecreted:

* Usage of Deprecated class or method or fields, shows you compile time warning.
* Deprecation means, no further improvements happening on this and use the alternative method or field instead.
* Can be used over: Constructor, Field, Local variable, method, Package, Parameter, Type(enum, class, interface ).

### @ Orderride:

* During compile time, it will check that the method should be overridden.
* Can be used over: methods.
* It throws compile time error if it doesn’t match with the parent method.

### @SupressWarnings

* This annotation will tell the compiler to ignore any compile time warnings.
* Use it safely, could run into runtime exception if, any valid warning is ignored.
* Can be used over: Field, Method, Parameter, Constructor, Local variable, Type(enum, class, interface ).

### @SafeVarargs:

* Used to suppress heap pollution warning. Heap pollution means object of one type, storing the reference of another type of object.
* Used on methods or constructors which has variable arguments as parameter.
* Methods should be static or final (i.e: methods which cannot be overridden).
* In java 9, we can use it on a private method too.

## Annotation used on another annotation(Meta annotation):

### @Target:

* This meta- annotation will restrict, where to use the other annotations. Either at method or constructor or fields, etc.
* using this annotation, we can control where other annotations work.
* To convert any annotation into a meta-annotation we can use the ElementType as Annotation\_Type .

### @Retention:

This annotation tells, how annotations will be stored in java.

RetentionPolicy.SOURCE:

Annotation will be discarded by the complier itself and it will not be recorded in the .class file.

RetentionPolicy. CLASS:

Annotation will be recorded in the .class file but will be ignore by the JVM at the runtime.

RetentionPolicy.RUNTIME:

Annotation will be recorded in the .class file+ available during the runtime. Usage of reflection can be done.

### @ Documented:

By default, Annotations are ignored when java Documentation is generated. With this meta-annotation even, annotations will come in java docs.

### @inherited:

By default, annotations applied on the parent class are not available to child class. If you use this annotation the child class inherits the annotations of the parent classes. This meta-annotation has no effect if annotation is used on other than a class.

### @Repeatable:

Allows us to use the same annotation more than 1 at same place.

We cannot do this before java 8.

## User-defined or Custom- annotation:

We can create our own annotations using the keyword “@interface”.

# Exception Handling:

* It’s an event, that occurs during the execution of the program.
* It disrupts the normal flow of the program.
* It creates the exception object, which contains information about the error like type of exception, message, and stack trace.
* Runtime system uses this exception object and find the class which can handle it. If it can’t find the class that can handle the exception, then runtime system will terminate the program abruptly and print the stack trace.

## Exception hierarchy:

### Throwable:

In Java, **Throwable** is the superclass of all errors and exceptions in the Java language. It serves as the base class for both checked exceptions (those that must be declared in a method's **throws** clause or handled with a **try-catch** block) and unchecked exceptions (those that don't need to be declared). The **Throwable** class provides methods to retrieve information about the exception, such as its message, cause, and stack trace.

**Throwable** has two main subclasses: **Error** and **Exception**. **Error** instances represent serious problems that typically cannot be recovered from, such as **OutOfMemoryError** or **StackOverflowError**, and are not meant to be caught or handled by application code. On the other hand, **Exception** instances represent exceptional conditions that can be handled by application code, such as **IOException** or **NullPointerException**.

### Error:

In Java, the **Error** class is a subclass of **Throwable** and represents serious, unchecked exceptions that typically indicate unrecoverable problems in the Java Virtual Machine (JVM) or the Java runtime environment. Errors are generally considered fatal and are not intended to be caught or handled by application code. Instead, they signal severe problems that may prevent the JVM from functioning properly or may indicate issues with the environment in which the Java program is running.

Some common subclasses of **Error** include:

1. **OutOfMemoryError**: Thrown when the JVM cannot allocate enough memory to perform an operation, such as creating a new object or loading a class.
2. **StackOverflowError**: Thrown when the JVM's stack space is exhausted, typically due to excessive recursion or deeply nested method calls.
3. **AssertionError**: Thrown when an assertion statement (assert) fails, indicating a logical error in the program's assumptions.
4. **ThreadDeath**: Thrown when a thread is forcibly terminated using the **Thread.stop()** method, though this method is deprecated and should not be used.

### Exception:

In Java, the **Exception** class is a subclass of **Throwable** and represents exceptional conditions that can occur during the execution of a program. Exceptions are used to signal errors, unexpected situations, or abnormal conditions that may arise while a program is running. Unlike errors, which typically indicate severe problems with the JVM or the runtime environment, exceptions are intended to be caught and handled by application code to gracefully recover from exceptional situations.

The **Exception** class itself is an abstract class, and it serves as the superclass for a wide range of exception types in Java. Subclasses of **Exception** represent specific types of exceptions, each with its own meaning and use cases.

## Unchecked/ Runtime Exception:

These are exception which occurs at the runtime and compiler does not force us to handle them.

**ClassCastException:**  
In Java, a ClassCastException is a runtime exception that occurs when an attempt is made to cast an object to a subclass type that is not compatible with its actual runtime type. This typically happens when there is a mismatch between the type of the object being cast and the type specified in the cast operation. For example, trying to cast an object of one class to another class that is not in its inheritance hierarchy will result in a ClassCastException.

**ArithmeticException:**

In Java, an **ArithmeticException** is a runtime exception that occurs when an arithmetic operation, such as division by zero, results in an invalid mathematical operation. This exception is typically thrown when attempting to perform a mathematical calculation that is not supported or results in an undefined or infinite value. For example, dividing an integer by zero or taking the remainder of a division operation where the divisor is zero will trigger an **ArithmeticException**. To prevent **ArithmeticException**s, it’s important to validate inputs and ensure that arithmetic operations are performed within the valid range of mathematical operations. When handling **ArithmeticException**s, developers can catch and handle the exception or implement defensive programming practices to avoid triggering arithmetic errors.

**IndexOutOfBoundsException:**

In Java, an IndexOutOfBoundsException is a runtime exception that occurs when attempting to access an element of an array, a collection, or a string using an index that is either negative or greater than or equal to the size or length of the data structure. This exception indicates that the index specified for accessing the element is out of the valid range of indices. For example, trying to access an element at index -1 or an index beyond the size of an array or list will trigger an IndexOutOfBoundsException. To prevent IndexOutOfBoundsExceptions, it's important to perform bounds checking before accessing elements by ensuring that the index is within the valid range of indices for the data structure. When handling IndexOutOfBoundsExceptions, developers can catch and handle the exception or implement defensive programming practices to validate indices before accessing elements.

**NullPointerException:**

In Java, a NullPointerException is a runtime exception that occurs when attempting to perform an operation on an object reference that is null, meaning it does not point to any object in memory. This exception is typically thrown when trying to access or invoke a method on a null reference, dereferencing a null pointer, or accessing an instance variable of an object that has not been initialized. For example, calling a method on a null object reference or trying to access a field of an object that has not been instantiated will result in a NullPointerException. To prevent NullPointerExceptions, it's important to ensure that object references are properly initialized before using them and perform null checks to handle potential null values gracefully. When handling NullPointerExceptions, developers can catch and handle the exception or implement defensive programming practices to avoid dereferencing null pointers.

## Checked/ Compile time exception:

Complier verifies them during the compile time of the code and if not handled properly, code compilation will fail.

Checked exceptions, also known as compile-time exceptions, are exceptions that must be declared in a method's throws clause or handled with a try-catch block at compile time. This means that the compiler checks for proper handling of checked exceptions during the compilation process, and if a checked exception is not handled or declared properly, a compilation error occurs. Examples of checked exceptions include **IOException**, **SQLException**, and **ClassNotFoundException**.

**Throws keyword:**

Throws keyword tells us that this method might throw exception (or might not ), so asks the caller method to handle it appropriately.

**Try/ catch block:**

In Java, the try-catch block is a mechanism used for exception handling, allowing developers to gracefully handle exceptions that may occur during the execution of code. The try block encloses the code that may potentially throw an exception, while the catch block handles any exceptions that are thrown within the try block. Catch blocks can only handle exceptions thrown by the try block.

Multiple catch blocks can be used to handle different types of exceptions, and a single try block can be followed by zero or more catch blocks.

Additionally, a finally block can be added after the catch block to specify code that should always be executed, regardless of whether an exception occurs or not. The finally block is commonly used for cleanup operations, such as closing resources or releasing locks.

**Finally keyword:**

* This block can be used after try or after catch block.
* This block will always get executed, either if you just return from try block or from catch block.
* At most, we can only add one finally block.
* Mostly used for closing the object, adding logs, etc.
* If JVM related issues like out of memory, system shutdown or our process is forcefully killed then finally block do not get executed.

**Throw keyword:**

This keyword is used to throw a new exception or to Re-throw the exception.

## Why we need exception handling?

* Exception handling makes our code clean by separating the error handling code from the regular code.
* It allows the program to recover from the error.
* It allows us to add information, which supports debugging.
* Improves security, by hiding the sensitive information.

## Disadvantages of exception handling:

* Exception handling is expensive, if stack trace is huge and it is handled or not handled at the parent class.

# Collections Framework:

Collections is nothing but group of objects. Its present in java.util package. This framework provides us the architecture to manage these “group of objects” i.e. add, update,delete,etc.

**Why we need Collection framework?**

Prior to this we have arrays, hash tables, vectors. But the problem with that is, there is no common interface, so it’s difficult to remember the methods of each data structure.

## Iterable interface:

In Java, the **Iterable** interface is a fundamental part of the Collections Framework, representing a collection of elements that can be iterated over sequentially. It provides a standard way to traverse the elements of a collection using the enhanced for loop (**for-each** loop) or by explicitly obtaining an iterator. Classes implementing the **Iterable** interface must implement the **iterator()** method, which returns an **Iterator** object allowing sequential access to the elements in the collection. This interface enables uniform traversal across various collection types, such as lists, sets, and maps, providing a common API for iterating over elements in Java collections.

Top of Form

## Collection interface:

In Java, the Collection interface is the root interface of the Java Collections Framework and represents a group of objects known as elements. It provides a common set of operations for working with collections, including adding, removing, and querying elements. Implementations of the Collection interface may vary in characteristics, such as allowing duplicate elements, maintaining order, or permitting null elements. Sub interfaces of Collection, such as List, Set, and Queue, further refine the behavior and properties of collections. The Collection interface defines methods such as add(), remove(), contains(), size(), and isEmpty(), among others, to manipulate and query the contents of a collection. It serves as the foundation for various data structures and algorithms in Java, enabling developers to work with collections in a standardized and flexible manner.

it represents the group of objects. It’s an interface which provides methods to work on group of objects.

## Queue Interface:

* Queue is an interface, child of collection interface.
* Generally, queue follows the FIFO approach, but there is exception like priority queue.
* Supports all the methods in the collection interface + some other methods defined in the queue interface.

## Priority queue:

* It’s of two types of minimum priority queue, maximum priority queue. It is based on priority heap (min heap, max heap). Elements are ordered according to their natural ordering (by default) or by comparator provided during queue construction time.
* By default, the priority queue is a min priority queue. To use the max priority queue, we need to implement the comparator along with the priority queue.
* Priority queue is not threading safe, the thread safe version is called PriorityBlockingQueue.
* Null elements not allowed.
* Insertion order is not maintained in the priority queue.
* Add and offer methods – O(logn)
* Peek – O(1)
* Poll and remove methods- O(logn)
* Remove arbitrary element – O(n)

## Comparator:

Sorting algorithm uses the compare method of the comparator to compare the two variables and decides whether to swap the variables or not. Comparator is a functional interface which has this compare method.

Method returns-

1 – if v1 is bigger than v2.

0 – if v1 is equal to v2

-1 – if v1 is less than v2.

Developers can implement the **Comparator** interface to create custom comparison logic tailored to specific requirements, such as sorting objects based on different attributes or in a different order than their natural ordering. Custom comparators are often used in conjunction with sorting algorithms provided by the **Collections** class, allowing objects to be sorted according to the defined comparison logic.

## Comparable:

In Java, the Comparable interface is used to define a natural ordering for objects of a class. It contains a single method, compareTo(), which compares the current object with another object of the same type and returns a negative integer, zero, or a positive integer depending on whether the current object is less than, equal to, or greater than the other object, respectively. Implementing the Comparable interface allows objects of a class to be compared and ordered, facilitating operations such as sorting and searching in collections. Classes that implement Comparable can be sorted using the Collections.sort() method.

## Deque Interface:

Deque stands for double ended queue. Means we can remove and add the elements from both the sides.

It has all methods of the queue interface like add, remove, poll, offer, peek, element.

**Methods present in the deque interface:**

A black screen with white text

Description automatically generated

Using these methods, we can even use deque to implement stack(LIFO) and Queue (FIFO).

To use it as stack, push() and pop() method are available.

Push() -> internally calls addFirst()

Pop() -> internally calls removeFirst().

## ArrayDeque:

ArrayDeque is a concrete class, and it implements the methods that are available in Queue and deque interfaces.

Time complexity:

Insertion- Amortized is O(1) except few cases like O(n) : when queue threshold reached and try to insert an element at the end or front then its O(n).

Deletion – O(1)

peeking- O(1)

Null elements not allowed.

ArrayDeque is not thread safe. The thread safe version is called concurrentLinkedDeque.

## List interface:

List is an ordered collection of an objects, in which duplicate values can be stored.

How does it differ from Queue?

Queue is also a collection of an objects but in queue insertion/ deletion/ access can only happen either at start or end while as in list the data can be inserted, removed, accessed from anywhere.

* List interface has all the methods in the collection interface + New methods defined in the list interface.
* List interface has a method called listIterator which helps in iterating in the backward direction.

## ArrayList:

ArrayList is based on array data structure. In Java, **ArrayList** is a dynamic array implementation of the **List** interface provided by the Java Collections Framework. It is a resizable array that can dynamically grow or shrink in size as elements are added or removed. Unlike traditional arrays, **ArrayList** automatically handles the resizing of the underlying array and provides methods to manipulate its contents.

ArrayList is not thread safe. The thread safe version is called CopyOnWriteArrayList.

It maintains insertion order.

In ArrayList we can insert null elements.

Time complexity in ArrayList:

Insertion:

O(1)- when inserting at the end of the ArrayList and space is sufficient.

O(n) – when inserting at index of the array, then it requires to shift the values.

O(n)- when array size threshold reached and try to insert an element at the end, then also its O(n) as values copied to new array of bigger size.

Deletion:

O(n) we must shift all the values.

Searching: O(1)

## LinkedList:

LinkedList implements both deque and list interface. Means it supports deque methods like removeFirst(), getFirst(), removeLast(), getLast(),etc.

It also supports index-based operations of the List interface.

Time complexity in LinkedList:

Insertion at the start or end is O(1).

Insertion at the particular index is O(n) for the lookup and O(1) for inserting.

Search – O(n)

Deletion at the start or the end O(1).

Deletion at a specific index O(n) for the lookup of the index, O(1) for the removal.

LinkedList is not thread safe.

Maintains the insertion order.

Null elements allowed.

## Vector:

Vector is exactly same as ArrayList, elements can access via index. But it is thread safe. Puts lock when an operation is performed on the vector. Less efficient than ArrayList as for each operation it does Lock and unlock internally. All the methods of the vector are synchronized methods. It maintains the insertion order and it allows the null elements.

## Stack:

Represents the LIFO (Last in First out ) operation.

Since it extends the vector, its methods are also synchronized. It is thread safe. We can implement the stack using the deque, but we know that deque is not thread safe.

Null elements are allowed.

It doesn’t maintain the insertion order.

Duplicate elements allowed.

Time complexity:

Insertion- O(1).

Deletion – O(1).

Search – O(n).

## Set Interface:

In Java, a **Set** is an interface provided by the Java Collections Framework that represents a collection of unique elements. Unlike a **List**, a **Set** does not allow duplicate elements; each element in a **Set** must be unique. Some common implementations of the **Set** interface in Java include **HashSet**, **TreeSet**, and **LinkedHashSet.** Set interface only has all the methods available in the collection interface. No new methods declared in set interface.

## HashSet:

In Java, **HashSet** is an implementation of the **Set** interface provided by the Java Collections Framework. It represents an unordered collection of unique elements, where each element is stored exactly once. **HashSet** uses a HashMap internally to achieve constant-time performance for basic operations such as adding, removing, and checking for the presence of elements. Sets cannot be accessed via index. HashSet do not guarantee the insertion order as the HashMap also don’t guarantee the insertion order. HashSet is not thread safe because HashMap is not thread safe. Null element is allowed but only once. To use the thread safe version, use the keyset method in the ConcurrentHashMap class.

As we know that HashSet uses HashMap internally. When we add an element to the HashSet it stores the element in the HashMap as key and creates a dummy as a value.

## LinkedHashSet:

LinkedHashSet Internally it uses the LinkedHashMap. It maintains the insertion order as the LinkedHashMap uses the double LinkedList. It’s not thread safe. Even though it uses LinkedHashMap internally LinkedHashSet Does not maintain the access order. The time complexity is same as LinkedHashMap which is amortized O(1).

## TreeSet:

Tree set uses the TreeMap internally. The time complexity is O(log(n)) as the TreeMap internally uses the red, black tree (Self balancing Binary search tree). We can use the comparator to manage the sorting order.

# Map interface:

Map is an interface, and its implementation are:

* HashMap – do maintain the order.
* HashTable – Synchronized version of HashMap.
* LinkedHashMap – Maintains the insertion order.
* TreeMap – Sorts the data internally.

Map helps to create objects that store key value pairs. in this key value pairs, we can’t have the duplicate keys.

## HashMap:

In Java, HashMap is an implementation of the Map interface provided by the Java Collections Framework. It stores key-value pairs and allows for efficient retrieval, insertion, and deletion of elements based on their keys. HashMap uses a hash table data structure to achieve constant-time performance for basic operations such as get(), put(), and remove() under normal circumstances. HashMap is not thread safe. Can store null key or null value. HashMap do not maintain the insertion order. To use the thread safe implementation, use HashTable or ConcurrentHashMap.

Average time complexity(Amortized) for searching inserting, deletion is O(1). Worst-case is O(n) if it uses LinkedList internally and O(log(n)) if it converts the LinkedList to Binary search tree. The conversion of LinkedList to Binary search tree happens after the LinkedList length reaches certain threshold.

**EntrySet() method** helps in iterating over the HashMap. We can use this method in the for-each loop (not for-each method )to iterate over each key-value pair. keyset() gives you all the keys of the HashMap. Values() methods gives you all the values in the HashMap.

## HashTable:

In Java, Hashtable is a legacy implementation of the Map interface provided by the Java Collections Framework. It stores key-value pairs and allows for efficient retrieval, insertion, and deletion of elements based on their keys. Hashtable is like HashMap but is synchronized, making it thread safe. While Hashtable provides thread safety, it comes at the cost of reduced performance due to synchronization overhead.

Null Keys and Values: Hashtable does not allow null keys or values. Attempting to insert a null key or value will result in a NullPointerException.

## LinkedHashMap:

In Java, **LinkedHashMap** is an implementation of the **Map** interface provided by the Java Collections Framework. It combines the features of a hash map with a doubly linked list, providing predictable iteration order while maintaining the fast retrieval and insertion capabilities.

LinkedHashMap maintains the insertion order and access order. It maintains the insertion order using the doubly LinkedList. Time complexity is same as HashMap. It’s not thread safe and there is no alternate version available. If we want a thread safe version we have to explicitly, do it using the collections.syncronized method.

Access order means the most accessed element will be moved to the last and least accessed will be in the First. To make the LinkedHashMap to behave this way we need to pass the access order as true while creating the LinkedHashMap.

## TreeMap:

The TreeMap in Java is used to implement [Map interface](https://www.geeksforgeeks.org/map-interface-java-examples/), Sorted interface and [NavigableMap](https://www.geeksforgeeks.org/navigablemap-interface-in-java-with-example/) along with the Abstract Map Class. The map is sorted according to the natural ordering of its keys, or by a [Comparator](https://www.geeksforgeeks.org/comparator-interface-java/) provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the TreeMap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. The TreeMap implementation is not synchronized in the sense that if a map is accessed by multiple threads, concurrently and at least one of the threads modifies the map structurally, it must be synchronized externally.

TreeMap is based on the red, black tree(Self balancing Binary search tree). O(log(n)) time complexity for the insert, remove, get operations.

# Streams:

In Java, streams are a powerful addition introduced in Java 8 as part of the Stream API, which is part of the Java Collections Framework. Streams provide a mechanism for processing sequences of elements in a functional style, enabling concise and expressive code for data manipulation and transformation. There are three steps to process data using the streams.

## Create Streams:

Fromcollections-Using the .stream() method.

From array – using the arrays.stream(List) method.

From Static method – using the stream.of( values) method.

From stream builder – first use the .builder() method and then use the .build method.

From Stream iterate – using the stream.iterate( logic here) method.

## Intermediate operations:

We can chain multiple intermediate operations to perform complex processing before applying the terminal operation to produce the result.

**Filter:** helps to filter the elements based on the logic defined as a lambda expression.

**Map**: it is used to transform each element using the logic defined in the lambda expression.

**FlatMap**: used to iterate over the complex collection and helps to flatten it as a stream.

**Distinct**: it removes the duplicates from the stream.

**Sorted**: this helps in sorting the elements.

**Peek**: helps to see the intermediate result of the stream which is getting processed. Using this we can print values in the intermediate stage.

**Limit**: Truncate the stream to have no longer than given max size.

**Skip**: using this we can first n elements in the stream.

**mapToInt**: helps to work with the primitive int data type.

mapToLong: helps to work with the primitive long data type in the streams.

mapToDouble: helps to work with the double data type in the streams.

**Why do we call intermediate operations lazy?**

Intermediate operations in streams are called lazy because they don't perform any processing until a terminal operation is called. This means that when you apply intermediate operations to a stream, such as filtering or mapping elements, the stream doesn't perform these operations right away. Instead, it builds a pipeline of operations to be executed.

Overall, the laziness of intermediate operations contributes to better performance, flexibility, and optimization in stream processing.

Top of Form

## Terminal operations:

These are the operations that triggers the stream and produces the result.

**forEach**: perform action on each element of the stream and do not return any value.

**toArray**: collects the elements of the stream into an array. By default, it returns an object array. If you want an array of certain objects like Integers, you need to implement your logic as a lambda expression inside the toArray method.

**reduce**: does reduction on the elements of the stream. Perform associative aggregation function. This function will return the Optional type of object.

**collect**: can be used to collect the elements of stream into a list.

**min**- finds the minimum or maximum element from the stream based on the comparator provided. This function will return the Optional type of object like optional<Integer>.

**max**- - finds the maximum or minimum element from the stream based on the comparator provided. This function will return the Optional type of object like optional<Integer>.

**count**: returns the count of elements present in the stream.

**anyMatch**: This operation checks if any value in the stream matches the given predicate and returns the Boolean.

**allMatch**: checks if all the values of the stream match the given predicate and returns the Boolean.

**noneMatch**: checks if no values in the stream match the given predicate and returns the Boolean.

**findFirst**: finds the first element of the stream. This returns the Optional type of object.

**findAny**: finds any random element of the stream. This returns the Optional type of object.

One terminal operation is used on a stream. It is closed/consumed and cannot be used again for another terminal operation.

## Parallel Stream:

Helps to perform operation on streams concurrently. Taking advantage of the multicore CPU.

parallelStream() method is used instead of regular stream() method.

Internally it does:

Task splitting: it uses “spliterator” functions to split the data into multiple chunks.

Task submission and parallel processing: Uses fork-join pool technique.

# Multithreading:

**process**

* process refers to an instance of a program that is being executed by the operating system.
* It has its own resource like memory, thread, etc.
* OS allocate these resources to the process.
* During the time of execution JVM starts the new process. Two process never share their resources.

**Thread**:

* Thread is known as lightweight process.
* Smallest sequence of instructions that are executed independently.
* One process can have multiple threads.
* When a process is created, it starts with one thread and that initial thread is known as main thread and from that we can create multiple threads to perform concurrently.
* Each thread gets its own stack, register, counter. They do not share to each other.
* They share the code segment, data segment and JVM instance heap memory.

**Code segment:**

Code segment contains the machine code of the java program. When we execute the process, the bytecode will be interpreted or complied to machine code by the JIT complier and stored in the code segment. All threads within the same process share the code segment. Threads can only read the code segment they can’t change it.

**Data Segment:**

Data segment contains the global and static variables. All threads within the same process, share the same data segment. Threads can read and modify the data. **Synchronization** is required in between the threads.

**Heap**:

Objects created during the runtime using the “new” keyword are stored in the heap.

Heap created inside the process is shared among all the threads. Threads can read the write to the heap data. **Synchronization** is required between threads.

**Stack**:

Each thread has its own stack. It manages method calls, local variables.

**Register**:

When the JIT (Just in time) complies convert the bytecode to machine code, it uses register to optimize the generated machine code. It also helps in context switching. Each thread has its own register.

**Counter**:

It’s also known as program counter. It points to the instruction which is getting executed by its own thread. Increments its counter after successful execution of the instruction.

**Steps of executing a program:**

* It will create a process.
* A new JVM instance is created inside the process.
* Each JVM Instance has its own heap memory, stack, code segment, Data segment, registers, program counters. We can tell how much heap memory need to be allocated per JVM instance or each process.
* The heap memory for each JVM instance will be allocated from the Total JVM Heap memory.
* Then JIT complier will start converting the bytecode to machine code. Machine will be saved into the code segment.
* Then threads are created, and machine code will get assigned to those threads for working on it. Counter will point to the instructions its working own.
* OS schedular will assign this thread to the CPU to run the code that belongs to the thread.
* After running for some time, the CPU will send the intermediate result back to thread register and then the CPU will work on another thread. This process of switching between threads by the CPU is called as **context switching**. It will again comeback and work on the first thread.

A screenshot of a computer

Description automatically generated

## Creating threads:

There are two ways to create a thread one is using the runnable interface and extending the thread class.

### Runnable interface:

A Runnable interface is commonly used in multi-threading to define tasks that can be executed concurrently. The Runnable interface has a single abstract method, **run()**, which is where the code for the task is defined. Here's how you can use the Runnable interface in multi-threading:

1. Implement the Runnable interface by creating a class that implements the **run()** method.
2. Instantiate an object of this class.
3. Pass the object to a Thread constructor.
4. Call the **start()** method on the Thread object to begin execution. This start() method will trigger the run method to work on its logic.

### Thread class:

In Java, the **Thread** class is a fundamental part of multithreading. It allows you to create and manage threads in your Java programs. You can create threads by subclassing **Thread** and overriding its **run()** method, or by passing a **Runnable** object to a **Thread** constructor.

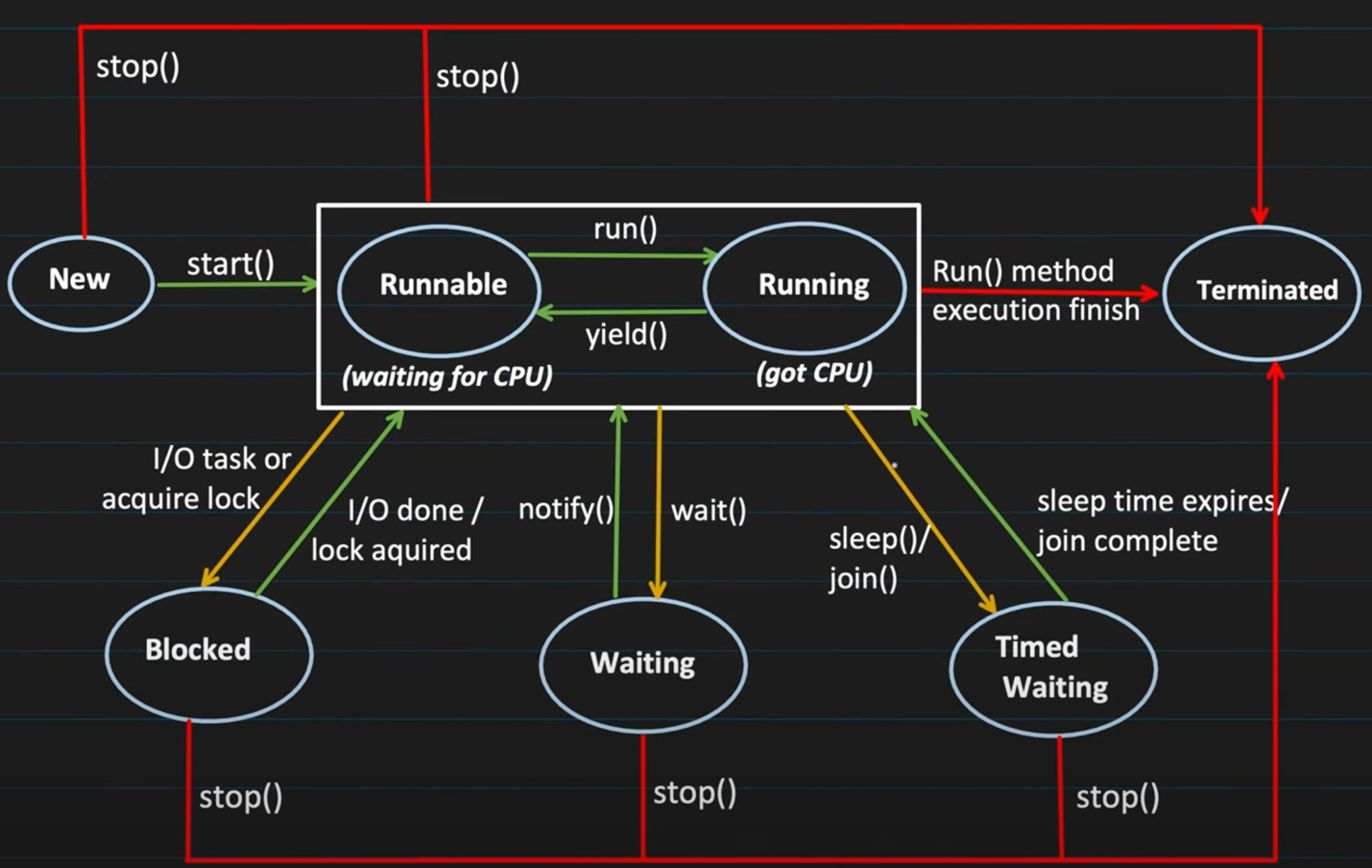
It's important to note that subclassing **Thread** tightly couples your code to the threading mechanism, while implementing **Runnable** provides better separation of concerns and allows for better object-oriented design. In most cases, implementing **Runnable** is preferred over subclassing **Thread**.

Here's how you can use the thread class in multi-threading via sub-classing :

1. Create a sub-class of thread class. Means create a class that extends the thread class. We need to override the run method here, as the parent run method doesn’t have any inbuilt logic.
2. Create the instance of the subclass. Call the start method on the sub class object to begin the execution. Top of Form

Bottom of Form

## Thread Life cycle:



**New:**

Thread has been but not yet started. It’s just an object in the memory.

**Runnable**:

Thread is ready to run. waiting for the CPU time.

**Running**:

When thread starts executing its code.

**Blocked**:

In different scenarios the thread will go into the blocked state.

* In I/O tasks like reading from a file or database.
* To acquire lock: if any thread wants to lock on a resource which is already locked by other thread, it must wait in the blocked state.

Whenever a thread goes to the blocked state it releases all the **monitor locks**.

**Waiting**:

Thread goes to this state whenever we call the wait() method, makes it non runnable. It goes back to runnable state only when we call notify() or notifyAll() method.

Whenever a thread goes to the waiting state it releases all the **monitor locks.**

**Timed waiting:**

Thread waits for specific period and comes back to the runnable state, after specific conditions met. Like sleep(), join().

Whenever a thread goes to the timed waiting state it does not releases any **monitor locks.**

**Terminated**:

Life of the thread is completed. It cannot be started back again.

## Monitor Lock:

“It helps to make sure only one thread goes inside the particular section of the code(synchronized block or method)”

A monitor lock, often simply referred to as a "lock," is a synchronization mechanism used in concurrent programming to control access to shared resources by multiple threads. In Java, every object has an associated monitor lock, which is used to ensure that only one thread can execute a synchronized block of code or method on that object at any given time. This prevents concurrent access to the shared resource, helping to avoid data corruption and race conditions.

**Why stop, resume, suspended methods deprecated?**

**Stop:** Terminates the thread abruptly, no lock release, no resource cleans up happens.

**Suspend**: put the thread on hold(suspend) for temporally. No lock release too.

**Resume**: used to resume the execution of suspended thread.

Both this suspend and stop operations could lead to issues like deadlocks.

## Join:

When join method is invoked on a thread object, current thread will be blocked and waits for the specific thread to finish.

It is helpful when we want to coordinate between threads or to ensure we complete certain tasks before moving.

## Thread priority:

Priorities are integers ranging from 1 to 10.

1 means low priority.

10 means high priority.

Even we set the thread priority while creation, it’s not guaranteed to follow any specific order, it’s just a hint to thread scheduler which to execute next. (but it’s not a strict rule).

When new thread is created, it inherits the priority of its parent thread.

We can set the custom priority using the setPriority(int priority).

## Daemon thread:

In Java, a daemon thread is a type of thread that runs in the background, providing services to non-daemon threads. Daemon threads are "low priority" threads because they don't prevent the JVM from exiting when all non-daemon threads have finished executing. In other words, if there are only daemon threads remaining, the JVM will terminate the program.

Here are some key characteristics of daemon threads:

Background Operations:

Daemon threads are typically used for tasks that need to run continuously in the background, such as garbage collection, monitoring, or logging.

Non-Blocking Exit:

If all non-daemon threads finish execution, the JVM will automatically terminate, regardless of whether daemon threads are still running. This behavior ensures that daemon threads do not prevent the JVM from shutting down.

Cleanup Tasks:

Daemon threads are often used to perform cleanup tasks or maintenance operations that should not prevent the program from exiting when it's done.

## Custom Locks:

Locks and semaphores are synchronization mechanisms used in concurrent programming to manage access to shared resources. A lock allows only one thread to access a resource at a time, ensuring mutual exclusion and preventing data corruption.

**These locks don’t depend on the objects like synchronize method.**

### Reentrant lock: ( the info is vague)

A reentrant lock, also known as a recursive lock, is a synchronization mechanism that allows a thread to acquire the same lock multiple times without causing a deadlock. In other words, a thread that already holds the lock can reacquire it without blocking itself.

The concept of reentrancy is particularly useful in scenarios where a method or function might need to acquire the same lock recursively. For example, consider a complex operation that involves multiple steps, each requiring exclusive access to a shared resource. If these steps are encapsulated within separate methods or functions, and each method or function needs to acquire the lock, reentrant locks ensure that a thread doesn't block itself when calling one of these methods recursively.

### ReadWrite lock:

Read lock can be acquired by more than one thread. Whereas the write can be acquired by only one thread. To take the write there should not be any other lock on the resource.

A read-write lock is a synchronization primitive that allows multiple readers to access a shared resource simultaneously, but it enforces exclusive access for a writer. This mechanism is particularly useful in scenarios where reads are more frequent than writes, and concurrent access by multiple readers does not risk data inconsistency.

### Stamped Lock:

A stamped lock is offering similar functionality to a read-write lock but with additional features. It supports three modes of locking: read, write, and optimistic read. In read mode, multiple threads can read concurrently. In write mode, exclusive access is granted to a single thread for writing. Optimistic read mode allows a thread to attempt to read optimistically without blocking, validating the stamped lock's state afterward. Stamped locks are versatile and can be useful in scenarios where reads significantly outnumber writes and where optimistic concurrency control is beneficial for performance optimization.

### Semaphore Lock:

semaphores can allow multiple threads to access a resource simultaneously, but within a **specified limit**. They maintain a count of available resources and control access based on this count, allowing threads to either wait or proceed depending on the availability. Both locks and semaphores are essential tools for coordinating concurrent activities and preventing race conditions in multi-threaded environments.

## LockFreeconcurrency:

### Compare and Swap:

(CAS) is an atomic instruction used in concurrent programming to implement lock-free algorithms and ensure synchronization between multiple threads. CAS allows a thread to update a variable's value if it matches an expected value. The operation consists of three inputs: a memory location (address), an expected value, and a new value. If the current value at the memory location matches the expected value, CAS atomically updates the value to the new value. CAS returns a Boolean indicating whether the operation was successful. If the value at the memory location has changed since the expected value was read, the operation fails, and the thread can retry or take alternative action. CAS is fundamental in building lock-free data structures and ensuring safe concurrent access to shared resources without the need for locks or mutexes, often leading to improved performance and scalability in multi-threaded applications.

Top of Form

### Atomic variables :

An **atomic operation** is an indivisible and uninterruptible operation that is guaranteed to be executed without interference from other concurrent operations. In concurrent programming, atomic operations are essential for ensuring thread safety and preventing data races.

Atomic variables are special types of variables used in concurrent programming to perform operations atomically, meaning they are executed as a single indivisible operation, without interference from other threads. Atomic variables are essential for ensuring thread safety and avoiding data races in multi-threaded environments.

Java provides a set of atomic variable classes in the **java.util.concurrent.atomic** package, such as **AtomicInteger**, **AtomicLong**, **AtomicBoolean**, etc.

Atomic variables have several advantages:

1. **Thread Safety**: Atomic variables ensure that operations performed on them are thread-safe, eliminating the need for explicit synchronization mechanisms like locks or semaphores.
2. **Performance**: Compared to using traditional locks, atomic variables often provide better performance in scenarios where contention is low, as they avoid the overhead of acquiring and releasing locks.
3. **Simple and Efficient**: Atomic variables are easy to use and understand, making concurrent programming less error prone. Additionally, they are efficient and scalable in multi-threaded environments.

**“**Atomic variables help in implementing lock free concurrency. Internally Compare and swap helps atomic variables to implement concurrency. Use the atomic variables when you do the read, modify, update operations in sequence, otherwise go with the standard locking. **”**

### Volatile variables:

Unlike other variables, *volatile* variables are written to and read from the main memory. **The CPU does not cache the value of a *volatile* variable.**

The *volatile* keyword is useful in two multi-threading scenarios:

* When only one thread writes to the *volatile* variable and other threads read its value. Thus, the reading threads see the latest value of the variable.
* When multiple threads are writing to a shared variable such that the operation is atomic. This means that the new value written does not depend on the previous value.

When Does volatile Not Provide Thread Safety?

Unlike *synchronized* methods or blocks, it does not make other threads wait while one thread is working on a critical section. Therefore, the *volatile* keyword does not provide thread safety **when non-atomic operations or composite operations are performed on shared variables**.

Operations like increment and decrement are composite operations. These operations internally involve three steps: reading the value of the variable, updating it, and then, writing the updated value back to memory.

The short time gap in between reading the value and writing the new value back to the memory can create a race condition. Other threads working on the same variable may read and operate on the older value during that time gap.

Moreover, if multiple threads are performing non-atomic operations on the same shared variable, they may overwrite each other’s results.

## Thread pools:

A thread pool is a collection of pre-initialized threads that are ready to perform tasks. Instead of creating a new thread for each task, a thread pool allows for the reuse of existing threads, which can significantly improve performance and reduce overhead in multi-threaded applications.

**Advantages of thread pool:**

1. Improved Performance:

Thread pools reduce the overhead associated with creating and destroying threads for each task. By reusing existing threads, the application avoids the cost of thread creation and context switching, leading to improved performance and reduced latency.

1. Resource Management:

Thread pools limit the number of concurrent threads, preventing resource exhaustion and overload. By controlling the number of threads, the application can manage system resources such as CPU and memory more efficiently, avoiding contention and ensuring smoother operation.

1. Concurrency Control:

Thread pools provide a centralized mechanism for controlling concurrency. By limiting the number of threads, they prevent excessive parallelism, which can lead to contention and decreased performance. They also manage task queuing, ensuring that tasks are processed in a controlled manner and preventing overload.

1. Thread Lifecycle Management:

Thread pools handle the lifecycle of threads, including creation, termination, and exception handling. This simplifies thread management for developers, as they don't need to deal with these details manually, reducing the risk of resource leaks and errors.

**A diagram of a service

Description automatically generated**

## ThreadPoolExecutor:

**ThreadPoolExecutor** is a class in Java's **java.util.concurrent** package that implements the **ExecutorService** interface, providing a flexible and configurable thread pool. It allows you to create and manage a pool of threads for executing tasks asynchronously. You can specify parameters such as the core pool size, maximum pool size, keep-alive time, and task queue type. It efficiently manages thread lifecycle, task queuing, and thread reuse, offering improved performance and resource management in multi-threaded applications.

Top of Form

### Bottom of FormParameters to create a ThreadPoolExecutor object:

**corePoolSize:**

Number of threads initially created and kept in the pool, even if they are idle.

**keepAliveTime**:

Thread which are idle gets terminated after this time. This happen only if the allowCoreThreadTimeOut property is set true.

**TimeUnit**:

This provides the time unit for the keepAliveTime, whether millisecond **or seconds or hours.**

**maxPoolSize**:

Maximum number of threads allowed in the pool.

If number of threads working == corePoolSize and queue is full, then new threads are created until the total number of threads not more than maxPoolSize.

**BlockingQueue**:

Queue used to hold task, before they got picked by the worker threads.

Bounded Queue: Queue with fixed capacity example: ArrayBlockingQueue.

Unbounded Queue: Queue with no fixed capacity example: LinkedBlockingQueue.

**ThreadFactory**:

Factory for creating new thread. ThreadPoolExecutor uses this to create new threads, this factory provides us an interface to:

* To give custom thread name.
* To give custom thread priority.
* To set thread daemon flag.

**RejectedExecutionHandler**:

Handler for tasks that cannot be accepted by the thread pool. Generally, logging logic can be put here. For debugging purposes. Java has inbuilt four rejection handler and we can also create a custom RejectedExecutionHandler**.**

### Lifecycle of ThreadPoolExecutor:

**Running State**:

When executor is in running state means, it can accept new tasks. Submit() method is used to add new task.

**Shutdown state:**

Executor do not accept new tasks, but continue to process on the existing tasks, once existing tasks finishes it will move to terminate state.

We can use the shutdown() method to perform this operation.

**Stop state (Force shutdown):**

Executor don’t accept new tasks. Executor forcefully stops all the tasks which are currently executing. Once fully shutdown it will be moved to terminate state. Methos used is shutdownNow().

**Terminated state**:

End of life for ThreadPoolExecutor. IsTerminated() method can be used to check whether a particular ThreadPoolExecutor is terminated or not.

### Future:

A **Future** in Java represents the result of an asynchronous computation. It provides a way to retrieve the result of an asynchronous operation, allowing the calling thread to continue execution while the operation is still in progress. The **java.util.concurrent.Future** interface was introduced in Java 5 as part of the concurrency utilities.

“Whenever we submit a task to a ThreadPoolExecutor object it returns a future object.”

It allows you to check if:

* Computation is complete.
* Get the result.
* Take care of exception if any.

Description of each method in the `java.util.concurrent.Future` interface:

1. boolean cancel(boolean mayInterruptIfRunning)`: Attempts to cancel the execution of the task associated with this `Future`.

2. get(): Waits if necessary for the computation to complete, and then retrieves its result.

3. get(long timeout, TimeUnit unit) : Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.

4. boolean isCancelled()`: Returns `true` if this task was cancelled before it completed normally.

5. boolean isDone()`: Returns `true` if this task completed.

### Callable:

submit() method can take both runnable and callable as the input for the thread.

Callable represents the task which need to be executed just like runnable.

But the difference is that runnable don’t have any return type. Whereas callable has the capability to return.

## CompletableFuture:

**CompletableFuture** is a class introduced in Java 8 as part of the CompletableFuture API, providing a more flexible and powerful way to work with asynchronous computations and concurrent programming compared to the traditional **Future** interface. It’s an advanced version of future.

It provides capabilities like chaining. This class implements the Future interface, so it has all the methods in the future interface, methods defined by itself.

**Methods of the CompletableFuture:**

### supplyAsync():

supplyAsync() is a static method in the CompletableFuture class introduced in Java 8. It is used to asynchronously execute a task that produces a result and returns a CompletableFuture representing that result. **supplyAsync()** allows you to execute a task asynchronously, by default using a separate thread from a thread pool managed by the **ForkJoinPool.commonPool()**. We can also provide the method with our own custom created executor. By passing the executor of our own we get more control over the thread.

**supplyAsync()** returns a **CompletableFuture<T>** representing the asynchronous result of the task. The future will be completed when the task finishes execution, either successfully or with an exception.

### thenApply():

**thenApply()** is a method in the **CompletableFuture** class introduced in Java 8. It is used to **chain** together asynchronous computations by applying a function to the result of a previous computation, returning a new **CompletableFuture** representing the result of the function.

allows you to specify a function that will be applied to the result of a previous computation, producing a new result asynchronously. It takes a **Function** as an argument, which represents the transformation to be applied to the result. The function takes the result of the previous computation as input and returns a new result of type. It’s a synchronous execution means it uses the same thread which completed the previous async task.

### thenApplyAsync():

thenApplyAsync() is a method in the CompletableFuture class introduced in Java 8. It is like thenApply(), but it applies a function asynchronously to the result of a previous computation, returning a new CompletableFuture representing the result of the function. It’s a Async execution means it uses different thread to complete the function.

If multiple thenApplyAsync are used ordering cannot be guaranteed they will run concurrently.

### thenCompose():

**thenCompose()** is a method in the **CompletableFuture** class introduced in Java 8. It is used to chain together asynchronous computations where the result of one computation depends on the result of another computation. It takes a **Function** as an argument that returns a **CompletableFuture**, and it returns a new **CompletableFuture** representing the result of the composed computation.

### thenComposeAsync():

**thenComposeAsync()** is a method in the **CompletableFuture** class introduced in Java 8. Similar to **thenCompose()**, it is used to chain together asynchronous computations where the result of one computation depends on the result of another computation. However, **thenComposeAsync()** applies the function using a different thread to the result of the previous computation, returning a new **CompletableFuture** representing the result of the composed computation.

### thenAccept(), thenAcceptAsync():

thenAccept() is a method in the CompletableFuture class introduced in Java 8. It is used to perform an action asynchronously when the computation represented by the CompletableFuture completes, without returning a result. **thenAccept()** returns a new **CompletableFuture<Void>** representing the completion of the action. The future will be completed when the action finishes execution.

Generally, it is used in the end stage, in the chain of the Async operations. It doesn’t return anything. thenAccept uses the same thread as previous operation whereas the thenAcceptAsync uses different thread.

### thenCombine, thenCombineAsync:

**thenCombine()** is a method in the **CompletableFuture** class introduced in Java 8. It is used to combine the results of two **CompletableFuture** instances when both have completed, applying a specified function to the results, and returning a new **CompletableFuture** representing the combined result.

thenCombine uses the same thread as previous operation whereas the thenCombineAsync uses different thread.

## Executors methods:

Java's Executors class provides factory methods for creating different types of ExecutorService instances. These factory methods simplify the process of creating thread pools with specific configurations. Executors provides factory methods which we can use to create ThreadPoolExecutor. Present in Java.util.concurrent package.

### FixedThreadPoolExecutor:

This method helps to create a thread pool executor with a fixed number of threads.

A black screen with white text

Description automatically generated

### CachedThreadPoolExecutor:

This method creates a thread pool that creates new thread as needed(Dynamically).

A screenshot of a computer program

Description automatically generated

### Single thread Executor:

This method creates executor with only one single worker thread.

A screenshot of a black screen

Description automatically generated

### work Stealing Pool:

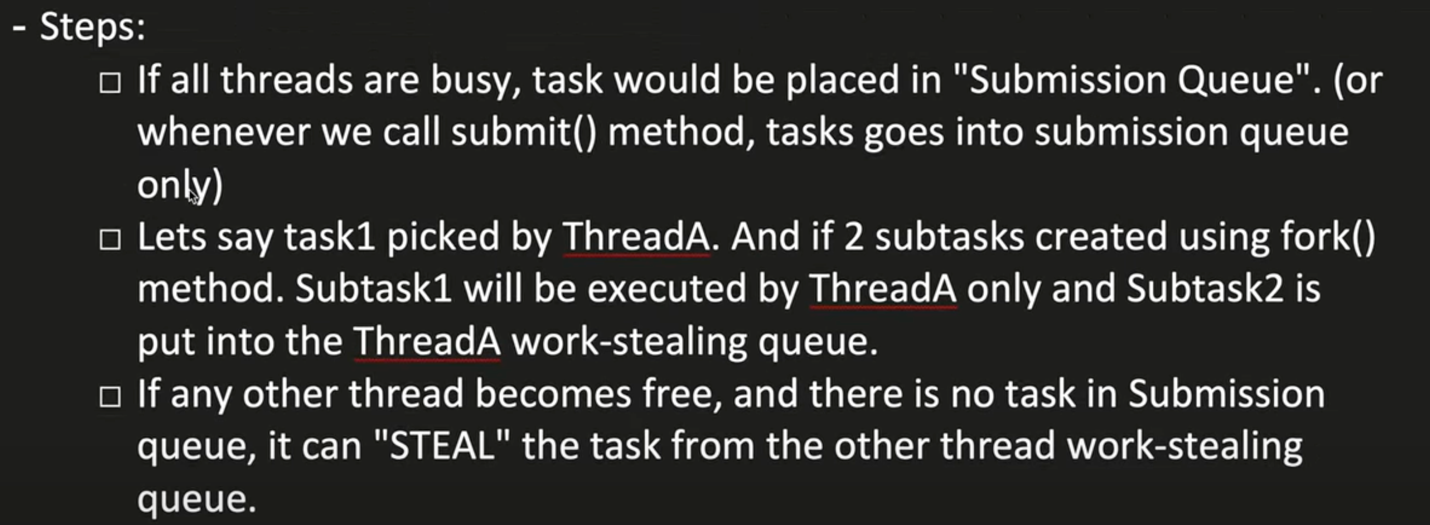
Creates a work stealing thread pool using all available processors as its target parallelism level. Tasks are distributed among worker threads to maximize parallelism.

It internally uses fork-join pool.

There are two types of queues:

1. Submission Queue
2. Work stealing queue for each thread (it’s a deque)

The work-stealing thread pool uses a work-stealing algorithm to distribute tasks among worker threads. In this algorithm, idle threads "steal" tasks from other threads' work stealing queues when their own work stealing queues, submission queue are empty. This approach helps to maximize CPU utilization and parallelism, especially in applications with irregular or dynamically changing workloads. We can custom provide how many threads to create, or it will create threads based on number of processors available in your computer.



Task can be split into multiple sub-tasks. For that task should extend:

* Recursive task – use this whenever your sub-task needs to return anything.
* Recursive Action- use this whenever your sub-task does not needs to return anything.

## Fork-Join Pool:

A ForkJoinPool is a special-purpose ExecutorService implementation in Java designed for recursive task execution using a work-stealing algorithm. It was introduced in Java 7 as part of the java.util.concurrent package and is particularly useful for parallelizing divide-and-conquer algorithms.

We can create Fork-join pool using “newWorkStealingPool” method in executor service or by calling ForkJoinPool.commonPool() method.

Here's how a ForkJoinPool typically works:

**Divide-and-Conquer**:

ForkJoinPool is designed to efficiently handle tasks that can be recursively divided into smaller subtasks, where each subtask is independent and can be executed in parallel. This pattern is known as divide-and-conquer.

**Forking**:

When a task is submitted to a ForkJoinPool, it is initially assigned to a "worker" thread for execution. If the task is deemed large enough, the worker thread may "fork" the task into smaller subtasks and distribute them among other worker threads in the pool for parallel execution.

**Joining**:

After forking the task, the worker thread may continue executing one of the subtasks while delegating other subtasks to other threads. Once a thread has completed its assigned subtask, it may "join" the results of the subtasks and continue with its execution.

**Work-Stealing**:

ForkJoinPool uses a work-stealing algorithm to achieve load balancing and efficient resource utilization. Each worker thread maintains its own double-ended queue (deque) of tasks. When a worker thread completes its own tasks, it may "steal" tasks from the tail of the deque of other worker threads. This helps to ensure that worker threads are kept busy and that the available processor cores are fully utilized.

**Task Granularity**:

To achieve optimal performance, it's important to strike a balance between the granularity of tasks and the overhead of task creation, forking, joining, and stealing. Tasks should be large enough to exploit parallelism effectively but not too large to incur excessive overhead.

**Parallelism Level**:

ForkJoinPool adapts to the available parallelism level of the underlying hardware, typically by creating one thread per processor core. However, you can also specify the parallelism level explicitly when creating a ForkJoinPool.

## ScheduledThreadPoolExecutor:

**ScheduledThreadPoolExecutor** is a class in Java that extends **ThreadPoolExecutor** and provides support for scheduling tasks to be executed after a certain delay, or to be executed periodically. It's part of the **java.util.concurrent** package and was introduced in Java 5 as part of the concurrency utilities.**it**  allows you to schedule tasks for execution after a specified delay or to be executed periodically at a fixed rate or with an initial delay.

**ScheduledThreadPoolExecutor** is commonly used in applications that require task scheduling and periodic execution, such as scheduling background tasks, performing periodic cleanup or maintenance tasks, and implementing timers or schedulers. It provides a flexible and efficient mechanism for managing scheduled tasks in multithreaded environments.

Top of Form

**methods in ScheduledThreadPoolExecutor:Bottom of Form**

schedule(Runnable command, long delay, TimeUnit unit):

Schedules a Runnable task for execution after the specified delay. Returns a ScheduledFuture representing the task.

schedule(Callable<V> callable, long delay, TimeUnit unit):

Schedules a Callable task for execution after the specified delay. Returns a ScheduledFuture<V> representing the task.

scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit): Schedules a Runnable task to be executed periodically with a fixed rate, beginning after the specified initial delay. Subsequent executions are scheduled with the same period, regardless of task execution duration.

scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit): Schedules a Runnable task to be executed periodically with a fixed delay between the completion of one execution and the start of the next execution. The initial delay specifies the delay before the first execution.

## Thread Local:

ThreadLocal is a class in Java that provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via it get() or set(Object) method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or transaction ID).

Here's how ThreadLocal works:

Independently Initialized Copies:

When you create a ThreadLocal instance and access its value using get() or set(Object), each thread that accesses the ThreadLocal will have its own, independently initialized copy of the variable.

Initialization:

When a ThreadLocal variable is first accessed via get() or set(Object) on a thread, it is typically initialized to a default value specified by the initialValue() method, which can be overridden.

Thread Safety:

ThreadLocal itself is thread safe. However, the objects stored within a ThreadLocal variable are not automatically thread safe. You must ensure that the objects stored in the ThreadLocal are either inherently thread-safe or properly synchronized if accessed concurrently by multiple threads.

Memory Leaks:

One important consideration when using ThreadLocal is the potential for memory leaks. Since each thread holds its own copy of the ThreadLocal variable, it's essential to ensure that resources associated with the ThreadLocal are properly cleaned up when they are no longer needed, typically by calling the remove() method.

Use Cases:

ThreadLocal is commonly used in multi-threaded applications where each thread requires its own instance of a variable with thread-specific data. Examples include storing user sessions in web applications, transaction context in database transactions, and per-thread logging contexts.

## Normal thread vs Virtual thread:

**Normal Threads**:

* Normal threads, also known as "native threads" or "OS threads", are managed by the operating system kernel.
* Each normal thread corresponds directly to an operating system thread, which can be resource-intensive to create and manage.
* Normal threads are heavyweight and have a relatively high memory overhead, especially when dealing with many threads.
* In Java, **Thread** class instances represent normal threads, and they are created using the **new Thread()** constructor or by implementing the **Runnable** interface.

**Virtual Threads**:

* Virtual threads, also known as "lightweight threads" or "green threads", are managed by the Java Virtual Machine (JVM) rather than the operating system.
* Virtual threads are lightweight compared to normal threads and have a much lower memory overhead.
* Multiple virtual threads can be multiplexed onto a smaller number of operating system threads, allowing for efficient use of system resources.
* Virtual threads are intended to be cheap to create and manage, making them suitable for applications that require many concurrent tasks.

In summary, normal threads are heavyweight, OS-managed threads with a high memory overhead, while virtual threads are lightweight, JVM-managed threads with low memory overhead, suitable for efficiently handling large numbers of concurrent tasks.

Top of Form

Bottom of Form