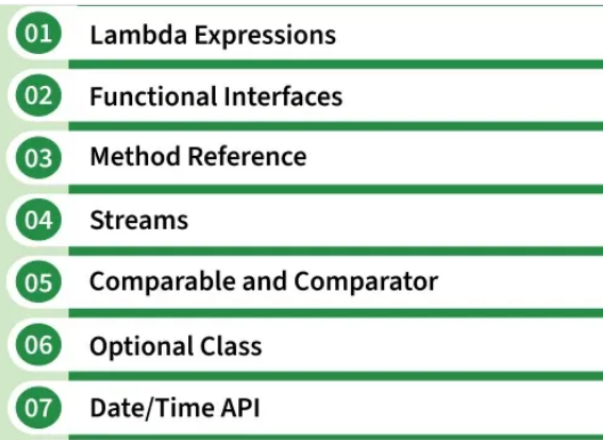
Java 8 Feature



Stream API

- Introduced in Java 8.
- Provides a functional programming approach to process collections of objects.
- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
- Streams are designed to be efficient and can support improving your program's performance by allowing you to avoid unnecessary loops and iterations. Streams can be used for filtering, collecting, printing, and converting from one data structure to another, etc.

Types of stream Operation

1. Intermediate Operations:

Intermediate operations transform a stream into another stream. Intermediate operations are lazy.

- filter(): Filters elements based on a specified condition.
- map(): Transforms each element in a stream to another value.
- sorted(): Sorts the elements of a stream.

2. Terminal Operations

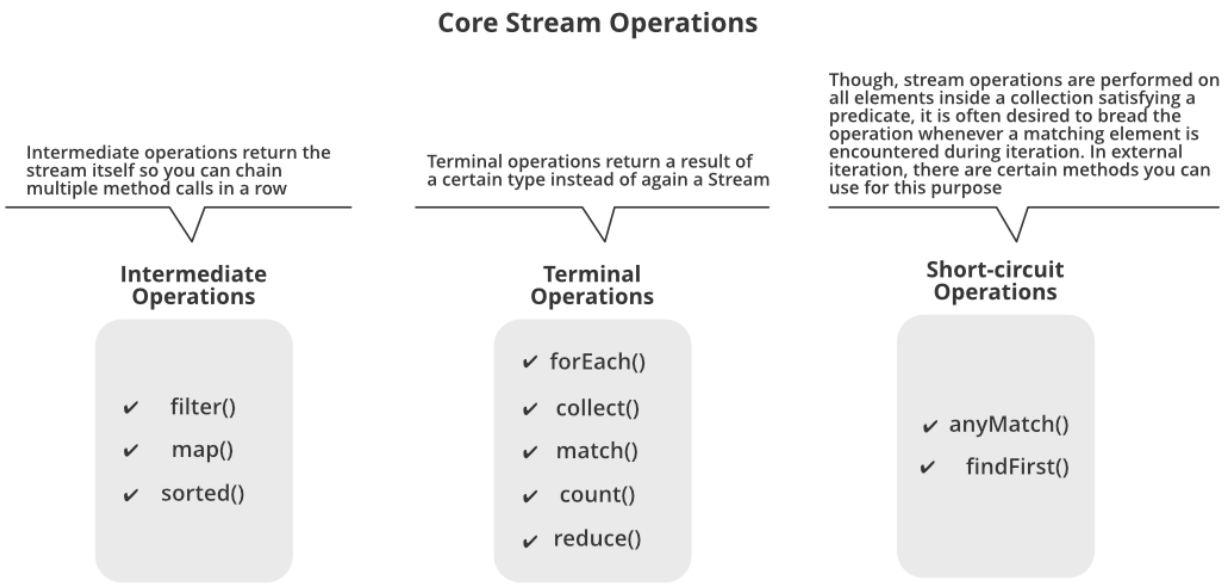
Terminal Operations are the operations that on execution return a final result as an absolute value. Terminal operations are eager.

- collect(): It is used to return the result of the intermediate operations performed on the stream.
- forEach(): It iterates all the elements in a stream.
- reduce(): It is used to reduce the elements of a stream to a single value.

3. Short Circuit Operations

Short-circuit operations provide performance benefits by avoiding unnecessary computations when the desired result can be obtained early. They are particularly useful when working with large or infinite streams.

- anyMatch(): it checks the stream if it satisfies the given condition.
- findFirst(): it checks the element that matches a given condition and stops processing when it finds it.



🔹 Key Features

1. Stream is not a data structure → it does not store data, it just processes it.

2. Lazy Evaluation → operations are executed only when a terminal operation is called.

3. Can be sequential or parallel.

4. Functional-style operations on data.

1.Printing only even numbers

```
List<Integer> list = Arrays.asList(4,5,9,1,2,0,3,7,8);
List<Integer> evenList = list.stream()
                    .filter(x->x%2==0)
                    .collect(Collectors.toList());
```

2.Change in uppercase

```
List<String> list = Arrays.asList("saroj", "shubham", "Suraj");
List<String> uperCaseList = list.stream()
                     .map(x->x.toUpperCase())
                     .collect(Collectors.toList());
```

3.Find first element greater than 10

```
List<Integer> list = Arrays.asList(5,8,9,20,30);
    Optional<Integer> first = list.stream()
                    .filter(x->x>10)
                    .findFirst();
```

4.Count frequency of words

```
List<String> list = Arrays.asList("Apple","Boy","Cat","Cat","Apple","Apple");
    Map<String, Long> freq = list.stream()
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

Functional Interface

A functional interface in Java is an interface that contains only one abstract method. Functional interfaces can have multiple default or static methods, but only one abstract method.

```
interface Square {
    int calculate(int x);
```

```
}
class Geeks {
    public static void main(String args[]) {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x) -> x * x;

        // parameter passed and return type must be same as defined
in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

@FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface cannot have more than one abstract method. In case more than one abstract methods are present, the compiler flags an "Unexpected @FunctionalInterface annotation" message. However, it is not mandatory to use this annotation.

Built-In Java Functional Interfaces

All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows:

Runnable: This interface only contains the **run()** method.

Comparable: This interface only contains the **compareTo()** method.

ActionListener: This interface only contains the **actionPerformed()** method.

Callable: This interface only contains the **call()** method.

Types of functional Interface

1. **Consumer:-** Consumer can be used in all contexts where an object needs to be consumed i.e taken as input, and some operation is to be performed on the object without returning any result. (void accept(T t))
2. **Predicate:-** This functional interface used for conditional check. (Boolean test (T t))

3. **Function:-** A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing. R apply(T t)
4. **Supplier:-** Supplier can be used in all contexts where there is no input but an output is expected. (T get())

T- generic data type

- **Function<T, R>**
    - Takes one argument of type T and returns a result of type R.
    - Example: Function<String, Integer> length = str -> str.length();
- **BiFunction<T, U, R>**
    - Takes two arguments of types T and U, returns a result of type R.
    - Example: BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
- **Consumer<T>**
    - Takes one argument of type T and returns nothing (void).
    - Example: Consumer<String> print = str -> System.out.println(str);
- **BiConsumer<T, U>**
    - Takes two arguments and returns nothing.
    - Example: BiConsumer<String, Integer> print = (s, i) -> System.out.println(s + i);
- **Supplier<T>**
    - Takes no arguments and returns a value of type T.
    - Example: Supplier<Double> random = () -> Math.random();
- **Predicate<T>**
    - Takes one argument and returns a boolean.
    - Example: Predicate<String> isEmpty = str -> str.isEmpty();
- **BiPredicate<T, U>**
    - Takes two arguments and returns a boolean.
    - Example: BiPredicate<String, String> equals = (a, b) -> a.equals(b);

🔹 **Unary and Binary Operators**

These are special cases of Function and BiFunction where input and output types are the same:

1. **UnaryOperator<T>**
    - Equivalent to Function<T, T>
    - Example: UnaryOperator<Integer> square = x -> x * x;
2. **BinaryOperator<T>**
    - Equivalent to BiFunction<T, T, T>
    - Example: BinaryOperator<Integer> multiply = (a, b) -> a * b;

Lamda Expression

- It is an anonymous function.
- A lambda expression can have 0 or any number of parameters.
- If body of lambda expressions contains only single line, then we can remove curly braces and return statement.
- It provides a clear and concise way to implement functional interface by using an expression.
- Lambda Expressions implement the only abstract function and therefore implement functional interfaces.

```
interface FuncInterface
{
    // An abstract function
    void abstractFun(int x);

    // A non-abstract (or default) function
    default void normalFun()
    {
       System.out.println("Hello");
    }
}

class Test
```

```
{
    public static void main(String args[])
    {
        // lambda expression to implement above functional
interface.
        FuncInterface fobj = (int x)->System.out.println(2*x);

        // This calls above lambda expression and prints 10.
        fobj.abstractFun(5);
    }
}
```

Structure of Lambda Expression

Types of Lambda Parameters

1. Lambda with Zero Parameter
2. Lambda with Single Parameter
3. Lambda with Multiple Parameters

Method Reference

A **method reference** is a **compact syntax** for writing lambda expressions that **invoke an existing method**.

They improve code readability and reduce boilerplate when you're simply passing a method as a parameter.

You can write: str -> str.toUpperCase()

You can write: String::toUpperCase

1. Static Method Reference

Syntax:

```
ClassName::staticMethodName
```

Example:

```
public class Utils {
    public static int square(int x) {
        return x * x;
    }
}
// Usage
Function<integer, integer> squareFunc = Utils::square;</integer, int
eger>
System.out.println(squareFunc.apply(5)); // Output: 25
```

2. Instance Method Reference of a Particular Object

Syntax:

```
instance::instanceMethodName
```

Example:

```
public class Printer {
    public void print(String msg) {
        System.out.println(msg);
    }
}
// Usage
Printer printer = new Printer();
Consumer printFunc = printer::print;
printFunc.accept("Hello, Java!"); // Output: Hello, Java!
```

3. Instance Method Reference of an Arbitrary Object of a Particular Type

Syntax:

```
ClassName::instanceMethodName
```

This is used when the method is called on the parameter passed to the lambda.

Example:

```
Function<string, integer> lengthFunc = String::length;</string, inte
ger>
System.out.println(lengthFunc.apply("Java")); // Output: 4
```

Here, String::length is equivalent to str -> str.length().

4. Constructor Reference

Syntax:

```
ClassName::new
```

Used to create new objects.

Example:

```
Supplier<list> listSupplier = ArrayList::new;</list
List list = listSupplier.get();
list.add("Java");
System.out.println(list); // Output: [Java]
```

Why Use Method References?

- **Cleaner syntax** than lambda expressions
- **Improves readability**
- **Avoids boilerplate code**

🔹 **Real-World Example with Streams**

```
List names = Arrays.asList("Alice", "Bob", "Charlie");
// Using lambda
names.forEach(name -> System.out.println(name));
// Using method reference
names.forEach(System.out::println);
```

Optional

The Optional class in Java is a **container object** used to represent the **presence or absence of a value**. It is used to avoid NullPointerException by providing methods to

check whether a value is present or not and handle it safely. By using Optional, we can specify alternate values to return or alternate code to run.

Method	Description
Optional.of(value)	Creates an Optional with a non-null value. Throws NullPointerException if value is null.
Optional.ofNullable(value)	Creates an Optional that may hold a null value.
Optional.empty()	Creates an empty Optional.
isPresent()	Returns true if a value is present.
ifPresent(Consumer)	Executes the given lambda if a value is present.
get()	Returns the value if present, else throws NoSuchElementException.
orElse(defaultValue)	Returns the value if present, else returns the default.
orElseGet(Supplier)	Returns the value if present, else gets it from a supplier.
orElseThrow()	Throws an exception if no value is present.

```
Optional name = Optional.ofNullable(getUserName());

name.ifPresent(n -> System.out.println("User name: " + n));

String defaultName = name.orElse("Guest");

System.out.println("Hello, " + defaultName);
```

Date & Time API

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :

1. **Not thread safe :** Unlike old java.util.Date which is not thread safe the new date-time API is *immutable* and doesn't have setter methods.

2. **Less operations :** In old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

1. **Local :** Simplified date-time API with no complexity of timezone handling.
2. **Zoned :** Specialized date-time API to deal with various timezones.

```
import java.time.*;

public class DateTimeExample {

    public static void main(String[] args) {

        LocalDate date = LocalDate.now();

        LocalTime time = LocalTime.now();

        LocalDateTime dateTime = LocalDateTime.now();

        ZonedDateTime zonedDateTime = ZonedDateTime.now();

        System.out.println("Date: " + date);

        System.out.println("Time: " + time);

        System.out.println("DateTime: " + dateTime);

        System.out.println("ZonedDateTime: " + zonedDateTime);

    }

}
```

Benefits of Java 8 Date and Time API

- Immutable and thread-safe
- Clear and fluent API design
- Better support for time zones and formatting
- Accurate calculations with Period and Duration

Comparable vs Comparator Interface

Comparable Interface

- **Package**: java.lang
- **Purpose**: Used to define the **natural ordering** of objects.
- **Method to implement**: compareTo(Object o)

- **Usage**: The class itself implements Comparable and overrides compareTo() to define how its instances should be compared.

```
public class Employee implements Comparable {
    private int id;
    public Employee(int id) {
        this.id = id;
    }
    @Override
    public int compareTo(Employee other) {
        return Integer.compare(this.id, other.id);
    }
}
```

Comparator Interface

- **Package**: java.util
- **Purpose**: Used to define **custom ordering** of objects, especially when you don't want to or can't modify the class.
- **Method to implement**: compare(Object o1, Object o2)
- **Usage**: Create a separate class or use a lambda to define comparison logic.

```
import java.util.Comparator;
public class EmployeeIdComparator implements Comparator {
    @Override
    public int compare(Employee e1, Employee e2) {
        return Integer.compare(e1.getId(), e2.getId());
    }
}
```

By Using lambda

```
// Sort by name using Comparator and lambda
employees.sort(Comparator.comparing(e -> e.getName()));po[
```

You can also use method references or Comparator.comparing() for cleaner code:

```
employees.sort(Comparator.comparing(Employee::getName));
```

Key Differences

Feature	Comparable	Comparator
Location	Implemented in the class itself	Separate class or lambda
Method	compareTo(T o)	compare(T o1, T o2)
Sorting Logic	Natural ordering	Custom ordering
Flexibility	Less flexible (one sort logic)	More flexible (multiple sort logics)
Use Case	When you control the class	When you don't control or need multiple orders

Note-> If below function return -1, then e1 should come before e2 b/c e1 is less than e2. If it returns 0 then they are equal and any one can comes at first. If it returns +1 then e2 will come before e1 b/c e2 is less than e1.

```
public int compare(Integer e1, Integer e2) {
        return e1-e2;
}
```

Different between String, String Buffer and String Builder

Summary of Differences

Feature	`String`	`StringBuffer`	`StringBuilder`
Mutability	Immutable	Mutable	Mutable
Thread-Safety	Thread-Safe	Thread-Safe (synchronized methods)	Not Thread-Safe (non-synchronized)
Performance	Lower performance in mutable operations due to object creation	Higher performance in mutable operations due to mutable nature	Higher performance in mutab operations due to non-synchronization
Memory Area	Heap Memory (String Pool)	Heap Memory (Internal mutable array)	Heap Memory (Internal mutable array)
Use Case	When string is constant or infrequent modifications	When string is modified in multi-threaded environment	When string is modified in single-threaded environment

Difference between Map and FlatMap

map() and **flatMap()** are both methods in Java's Stream API, but they serve different purposes.

1. map()

- **Purpose**: Transforms each element of a stream into another element.
- **Output**: Produces a **one-to-one mapping** (i.e., for each input element, there is exactly one output element).
- **Use Case**: When you want to apply a function to each element of a stream and get a transformed stream.

Example:

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<Integer> nameLengths = names.stream()
                                 .map(String::length)
                                 .collect(Collectors.toList());
System.out.println(nameLengths); // Output: [5, 3, 7]
```

Here, map() transforms each name into its length.

2. flatMap()

- **Purpose**: Transforms each element into a stream and then **flattens** the resulting streams into a single stream.

- **Output**: Produces a **one-to-many mapping** (i.e., for each input element, there can be zero, one, or multiple output elements).
- **Use Case**: When you have nested structures (e.g., Stream<Stream<T>>) and want to flatten them into a single stream.

Example:

```
List<List<String>> nestedList = Arrays.asList(
    Arrays.asList("Alice", "Bob"),
    Arrays.asList("Charlie", "David"),
    Arrays.asList("Eve")
);
List<String> flatList = nestedList.stream()
                                  .flatMap(List::stream)
                                  .collect(Collectors.toList());
System.out.println(flatList); // Output: [Alice, Bob, Charlie, David, Eve]
```

OR

```
List<String> flatList = nestedList.stream()
                                  .flatMap(innerList -> innerList.stream())
                                  .collect(Collectors.toList());
```

Here, flatMap() flattens the nested lists into a single list.

Key Difference

- **map()**: Transforms each element individually.
- **flatMap()**: Transforms each element into a stream and merges all resulting streams into one.

Example

- Use map to get a list of each customer's orders.
- Use flatMap to get a single list of all orders across all customers.

Difference between HashMap, HashTable and ConcurrentHashMap

🗂️ 1. HashMap

- **Thread-safe?** ❌ No
- **Allows null keys/values?** ✅ One null key and multiple null values
- **Performance:** Fast in single-threaded environments
- **Synchronization:** Must be manually synchronized if used in multi-threaded code
- **Introduced in:** Java 1.2

✅ Use when:

- You're working in a **single-threaded** context
- You need **null keys or values**
- You want **better performance** without thread safety

🧱 2. Hashtable

- **Thread-safe?** ✅ Yes (synchronized methods)
- **Allows null keys/values?** ❌ No
- **Performance:** Slower due to synchronization overhead
- **Synchronization:** Every method is synchronized
- **Introduced in:** Java 1.0 (legacy class)

❌ Avoid unless:

- You're maintaining **legacy code**
- You need **built-in synchronization** and don't mind the performance hit

⚙️ 3. ConcurrentHashMap

- **Thread-safe?** ✅ Yes (uses internal partitioning)
- **Allows null keys/values?** ❌ No
- **Performance:** High performance in concurrent environments
- **Synchronization:** Uses **lock striping** (segments of the map are locked independently)
- **Introduced in:** Java 1.5

✅ Use when:

- You're working in a **multi-threaded** environment
- You need **high concurrency** and **better performance**
- You don't need to store null keys or values

Difference between stream and parallel stream

In Java (and other languages with similar concurrency models), **stream()** and **parallelStream()** are used to process collections of data, but they differ in how they execute:

🔹 **stream()**

- **Sequential processing**: Elements are processed one after another.
- **Single-threaded**: Uses the main thread or a single thread from the ForkJoinPool.
- **Predictable order**: Maintains the order of elements as they appear in the source.
- **Use case**: Best for small datasets or when order matters.

🔹 **parallelStream()**

- **Parallel processing**: Elements are divided into chunks and processed concurrently.
- **Multi-threaded**: Uses multiple threads from the common ForkJoinPool.
- **Less predictable order**: May not preserve the order of elements.
- **Use case**: Suitable for large datasets or CPU-intensive operations where parallelism improves performance.

⚠️ **Considerations**

- **Performance gain** is not guaranteed with parallelStream()—it depends on the task, data size, and system resources.
- **Thread-safety**: Be cautious when using shared mutable data.
- **Overhead**: Parallel streams introduce overhead due to thread management.

✅ **Example**

```
List names = Arrays.asList("Alice", "Bob", "Charlie");
// Sequential stream
names.stream().forEach(System.out::println);
// Parallel stream
names.parallelStream().forEach(System.out::println);
```

Difference between abstraction & encapsulation

Abstraction

✅ **Definition:**

Abstraction is the process of **hiding the implementation details** and showing only the **essential features** of an object.

✅ **Purpose:**

To **simplify complexity** by exposing only relevant data and operations.

✅ **How it's achieved in Java:**

- Using **abstract classes** and **interfaces**

Example:

```
interface Animal {
    void makeSound(); // abstract method
}
class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

✅ **Real-world analogy:**

When you drive a car, you use the steering wheel and pedals without knowing how the engine works internally.

🔹 Encapsulation

✅ **Definition:**

Encapsulation is the process of **wrapping data (variables)** and **code (methods)** together into a single unit, and **restricting access** to some of the object's components.

✅ **Purpose:**

To **protect data** from unauthorized access and modification.

✅ **How it's achieved in Java:**

- Using **private fields** and **public getters/setters**

Example:

```
class Person {
    private String name; // private field
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

✅ **Real-world analogy:**

Think of a capsule — it hides the medicine inside and controls how it's released.

🔸 **Key Differences**

Feature	Abstraction	Encapsulation
Focus	Hiding implementation details	Hiding internal state/data
Purpose	Simplify usage	Protect and control access
Achieved via	Abstract classes, interfaces	Access modifiers (private, public)
User knows	What the object does	How to interact with the object safely
Example	Animal.makeSound()	Person.getName() / setName()

Types of Polymorphism in Java

1. **Compile-time Polymorphism** (also called **Static Polymorphism**)
   Achieved through **method overloading**.
2. **Runtime Polymorphism** (also called **Dynamic Polymorphism**)
   Achieved through **method overriding**.

🔹 **Method Overloading vs. Method Overriding**

Feature	Method Overloading	Method Overriding
Definition	Same method name, different parameters	Same method name and parameters in subclass
Type	Compile-time polymorphism	Runtime polymorphism
Inheritance Required?	❌ No	✅ Yes
Return Type	Can be different	Must be same or covariant
Access Modifier	Can be anything	Cannot reduce visibility
Static Methods	Can be overloaded	Cannot be overridden

What is the role of interfaces in Java OOP? How are they different from abstract classes?

In Java's object-oriented programming (OOP) model, **interfaces** and **abstract classes** are both used to achieve **abstraction**, but they serve different purposes and have distinct characteristics.

🔹 **Role of Interfaces in Java OOP**

✅ **Definition:**

An **interface** in Java is a blueprint for a class that defines a set of abstract methods and constants. It can contain **abstract methods**, **default methods**, **static methods**, and **constants**.

✅ **Purpose:**

- Define a **contract** that classes must follow.
- Enable **multiple inheritance of behavior**.
- Promote **loose coupling** and **polymorphism**.

✅ **Example:**

```
interface Animal {
    void makeSound(); // abstract method
}
class Dog implements Animal {
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

Here, Dog implements the Animal interface and provides its own version of makeSound().

🔹 **Key Features of Interfaces**

- All methods are **implicitly public and abstract** (unless marked default or static).
- Fields are **public, static, and final** by default.
- A class can **implement multiple interfaces**.
- Interfaces support **default** and **static methods** (since Java 8).
- Interfaces can be **functional interfaces** (with a single abstract method) for use with **lambda expressions**.

🔹 **Abstract Classes in Java**

✅ **Definition:**

An **abstract class** is a class that cannot be instantiated and may contain **abstract methods** (without implementation) and **concrete methods** (with implementation).

✅ **Example:**

```
abstract class Animal {
    abstract void makeSound(); // abstract method
    void breathe() {
        System.out.println("Breathing...");
    }
}
class Cat extends Animal {
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

🔹 **Differences Between Interface and Abstract Class**

Feature	Interface	Abstract Class
Inheritance	Multiple inheritance allowed	Single inheritance only
Method Types	Abstract, default, static	Abstract and concrete
Field Types	public static final only	Any type (private, protected, etc.)
Constructor	No constructors	Can have constructors
Access Modifiers	Methods are public by default	Can use any access modifier
Use Case	Define behavior contracts	Provide partial implementation
Instantiation	Cannot be instantiated	Cannot be instantiated

🔹 **When to Use What?**

- Use an **interface** when:
    - You want to define a **contract** for unrelated classes.
    - You need **multiple inheritance** of behavior.
    - You're working with **functional programming** (e.g., lambda expressions).
- Use an **abstract class** when:
    - You want to provide a **base class** with some shared implementation.
    - You expect classes to be **closely related**.
    - You need **constructors**, **non-final fields**, or **non-public methods**.

In Java, super is a keyword that lets a subclass interact with its **immediate superclass**. It's primarily used to:

1. **Call a superclass constructor**
2. **Access a hidden field** (same name in subclass and superclass)
3. **Invoke a superclass method** that's been overridden in the subclass
4. **(Generics)** Specify a **lower-bounded wildcard** (e.g., List<? super Integer>)

Below are the details, with examples and rules you'll actually use.

What is the significance of the super keyword in Java?

1) Calling the superclass constructor

Every constructor must start by calling either this(...) (another constructor in the same class) or super(...) (a constructor in the parent). If you don't write one, the compiler inserts super();**only if** the parent has a no-arg constructor.

```
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
}
```

```
class Employee extends Person {
    int id;
    Employee(String name, int id) {
        super(name);   // must be the first statement
        this.id = id;
    }
}
```

Rules

- super(…) must be the **first** statement in a constructor.
- You can call **either** this(…) **or** super(…), not both.
- If the superclass has **no no-arg constructor**, you **must** call an existing one explicitly.

2) Accessing a hidden field

Fields aren't overridden—**they're hidden** when the subclass declares a field with the same name. Use super.fieldName to access the parent's version.

```
class A {
    int x = 10;
}
class B extends A {
    int x = 20;
    int sum() {
        return x + super.x;  // 20 + 10 = 30
    }
}
```

3) Invoking an overridden method in the superclass

When a subclass overrides a method, you can still call the parent version using super.methodName(…).

```
class Logger {
    void log(String msg) { System.out.println("[LOG] " + msg); }
```

```
}
class TimestampLogger extends Logger {
    @Override
    void log(String msg) {
        super.log(java.time.Instant.now() + " " + msg);
    }
}
```

Notes

- This is helpful when you want to extend (not replace) parent behavior.
- You **cannot** use super to skip multiple levels (no super.super).
- super resolves **at compile time** to the immediate parent's implementation.

4) Using super with default methods from interfaces

If multiple interfaces provide default methods with the same signature, you must resolve the conflict. Inside your override, you can call a specific interface's default method with InterfaceName.super.method().

```
interface A { default void hello() { System.out.println("A"); } }
interface B { default void hello() { System.out.println("B"); } }
class C implements A, B {
    @Override
    public void hello() {
        A.super.hello();  // choose A's default
        B.super.hello();  // and also call B's default
    }
}
```

Important rules & caveats

- super **cannot** be used in a **static** context (no super in static methods/blocks).
- You can't access **private** members of the superclass with super (privacy rules still apply).
- You can write super.toString() etc., but you **cannot** use super to call static members (static members belong to the class, not the instance).

- **Qualified super** in inner classes: from an inner class, you can target the superclass of the **enclosing instance**: OuterClass.super.method().
- No super.super in Java.

Quick comparison: this vs super

- this → current class's instance (call peer constructors, refer to current fields/methods).
- super → current instance **as viewed by** the immediate superclass (call parent constructor, parent methods/fields).

How do you ensure code reusability and maintainability in your OOP design?

Ensuring **code reusability** and **maintainability** in Object-Oriented Programming (OOP) design involves applying several principles and best practices. Here's a breakdown of how to achieve both:

✅ **Code Reusability**

1. **Encapsulation**
    - Keep data and behavior bundled together in classes.
    - Hide internal implementation details using access modifiers (private, protected, public).
2. **Inheritance**
    - Create base classes with common functionality.
    - Allow derived classes to reuse and extend behavior.
3. **Polymorphism**
    - Use interfaces or abstract classes to define common behavior.
    - Enable different implementations to be used interchangeably.
4. **Composition Over Inheritance**
    - Prefer composing objects with desired functionality rather than inheriting from a base class.
    - This makes code more flexible and reusable.

5. **Design Patterns**
    - Apply reusable solutions like Factory, Strategy, Observer, etc., to common problems.
6. **Modular Design**
    - Break code into small, independent modules or classes that can be reused across projects.

🔧 **Code Maintainability**

1. **SOLID Principles**
    - **S**ingle Responsibility: Each class should have one reason to change.
    - **O**pen/Closed: Classes should be open for extension but closed for modification.
    - **L**iskov Substitution: Subtypes must be substitutable for their base types.
    - **I**nterface Segregation: Prefer many small interfaces over one large one.
    - **D**ependency Inversion: Depend on abstractions, not concretions.
2. **Clean Code Practices**
    - Use meaningful names.
    - Keep methods short and focused.
    - Avoid deep nesting and complex logic.
3. **Documentation & Comments**
    - Document public APIs and complex logic.
    - Avoid redundant or misleading comments.
4. **Unit Testing**
    - Write tests to ensure code behaves as expected.
    - Makes refactoring safer and easier.
5. **Refactoring**
    - Regularly improve code structure without changing behavior.
    - Helps eliminate duplication and improve readability.
6. **Version Control & Code Reviews**

- Use Git or similar tools to track changes.
- Peer reviews help catch issues early and share knowledge.

How do you handle tight coupling between classes?

Tight coupling means classes are highly dependent on each other, making the system harder to maintain, test, and extend. To handle or reduce tight coupling, you can apply these strategies:

✅ **1. Use Interfaces or Abstract Classes**

Instead of depending on concrete classes, depend on abstractions (**Dependency Inversion Principle**).

✅ **2. Apply Dependency Injection (DI)**

Inject dependencies instead of creating them inside the class. This can be done via:

- **Constructor Injection**
- **Setter Injection**
- **Frameworks** like Spring for IoC (Inversion of Control)

✅ **3. Use Design Patterns**

- **Factory Pattern**: To create objects without exposing creation logic.
- **Strategy Pattern**: To switch algorithms at runtime.
- **Observer Pattern**: To decouple event handling.

✅ **4. Favor Composition over Inheritance**

Instead of extending classes (which creates strong coupling), use composition to include behavior.

✅ **5. Apply SOLID Principles**

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

✅ **6. Use Event-Driven or Messaging Systems**

For large systems, use **publish-subscribe** or **message queues** (e.g., Kafka, RabbitMQ) to decouple components.

What are SOLID principles?

The SOLID principles are five essential guidelines that enhance software design, making code more maintainable and scalable.

- The SOLID principles help in enhancing loose coupling. Loose coupling means a group of classes are less dependent on one another.
- Loose coupling helps in making code more reusable, maintainable, flexible and stable. Now let's discuss one by one these principles...
- Loosely coupled classes minimize changes in your code when some changes are required in some other code.

S : Single Responsibility Principle (SRP)

O : Open closed Principle (OSP)

L : Liskov substitution Principle (LSP)

I : Interface Segregation Principle (ISP)

D : Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP):-

This principle states that "A class should have only one reason to change" which means every class should have a single responsibility or single job or single purpose.

Example: Imagine a baker who is responsible for baking bread. The baker's role is to focus on the task of baking bread, ensuring that the bread is of high quality, properly baked, and meets the bakery's standards.

- However, if the baker is also responsible for managing the inventory, ordering supplies, serving customers, and cleaning the bakery, this would violate the SRP.
- Each of these tasks represents a separate responsibility, and by combining them, the baker's focus and effectiveness in baking bread could be compromised.
- To adhere to the SRP, the bakery could assign different roles to different individuals or teams. For example, there could be a separate person or team responsible for managing the inventory, another for ordering supplies, another for serving customers, and another for cleaning the bakery.

```
// Class for baking bread
class BreadBaker {
    public void bakeBread() {
        System.out.println("Baking high-quality bread...");
    }
}

// Class for managing inventory
class InventoryManager {
    public void manageInventory() {
        System.out.println("Managing inventory...");
    }
}

// Class for ordering supplies
class SupplyOrder {
    public void orderSupplies() {
        System.out.println("Ordering supplies...");
    }
}

// Class for serving customers
class CustomerService {
    public void serveCustomer() {
        System.out.println("Serving customers...");
    }
}

// Class for cleaning the bakery
```

```
class BakeryCleaner {
    public void cleanBakery() {
        System.out.println("Cleaning the bakery...");
    }
}

public class Main {
    public static void main(String[] args) {
        BreadBaker baker = new BreadBaker();
        InventoryManager inventoryManager = new InventoryManager();
        SupplyOrder supplyOrder = new SupplyOrder();
        CustomerService customerService = new CustomerService();
        BakeryCleaner cleaner = new BakeryCleaner();

        // Each class focuses on its specific responsibility
        baker.bakeBread();
        inventoryManager.manageInventory();
        supplyOrder.orderSupplies();
        customerService.serveCustomer();
        cleaner.cleanBakery();
    }
}
```

Open closed Principle (OSP):-

This principle states that "**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**" which means you should be able to extend a class behavior, without modifying it.

This can be achieved by using Polymorphism, Inheritance, Abstraction.

Example: Imagine you have a class called PaymentProcessor that processes payments for an online store. Initially, the PaymentProcessor class only supports processing

payments using credit cards. However, you want to extend its functionality to also support processing payments using PayPal.

Instead of modifying the existing PaymentProcessor class to add PayPal support, you can create a new class called PayPalPaymentProcessor that extends the PaymentProcessor class. This way, the PaymentProcessor class remains closed for modification but open for extension, adhering to the Open-Closed Principle.

```
// Base class for payment processing
abstract class PaymentProcessor {
    public abstract void processPayment(double amount); // Pure virtual function
}

// Credit card payment processor
class CreditCardPaymentProcessor extends PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of $" + amount);
    }
}
```

Extended Functionality: Now, to add support for PayPal payments, you create a new class PayPalPaymentProcessor that extends PaymentProcessor.

```
// PayPal payment processor
class PayPalPaymentProcessor extends PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of $" + amount);
    }
}
```

Usage:

In your application, you can use either payment processor without modifying the existing code for PaymentProcessor or CreditCardPaymentProcessor.

```
public class Main {

    public static void main(String[] args) {

        CreditCardPaymentProcessor creditCardProcessor = new
CreditCardPaymentProcessor();

        PayPalPaymentProcessor payPalProcessor = new
PayPalPaymentProcessor();


        processPayment(creditCardProcessor, 100.00); // Processing
credit card payment

        processPayment(payPalProcessor, 150.00);     // Processing
PayPal payment

    }


    // Function to process payment

    public static void processPayment(PaymentProcessor processor,
double amount) {

        processor.processPayment(amount);

    }

}
```

Liskov substitution Principle (LSP):-

The principle was introduced by Barbara Liskov according to this principle "**Derived or child classes must be substitutable for their base or parent classes**".

Example:

❌ **A Violating Example (What *not* to do)**

We design a base Account with withdraw, and a subtype FixedDepositAccount that *throws* on withdraw (because fixed deposits usually don't support withdrawals before maturity). Clients using Account will break when substituted with FixedDepositAccount.

```
abstract class Account {

    protected BigDecimal balance = BigDecimal.ZERO;
```

```
    public void deposit(BigDecimal amount) {
        if (amount.signum() <= 0) throw new IllegalArgumentException
("Amount must be positive");
        balance = balance.add(amount);
    }
    // Base contract (implicit): withdraw should succeed if funds ar
e sufficient.
    public void withdraw(BigDecimal amount) {
        if (amount.signum() <= 0) throw new IllegalArgumentException
("Amount must be positive");
        if (balance.compareTo(amount) < 0) throw new IllegalStateExc
eption("Insufficient funds");
        balance = balance.subtract(amount);
    }
    public BigDecimal getBalance() {
        return balance;
    }
}
class CheckingAccount extends Account {
    // Inherits behavior; fine.
}
class FixedDepositAccount extends Account {
    // ❌ Violates LSP: strengthens preconditions by forbidding wit
hdrawals altogether.
    @Override
    public void withdraw(BigDecimal amount) {
        throw new UnsupportedOperationException("Withdrawals are not
 allowed before maturity");
    }
}
class RefundService {
```

```
    // A client that relies on the Account contract:
    public void issueRefund(Account account, BigDecimal amount) {
        account.withdraw(amount); // Works for CheckingAccount, cras
hes for FixedDepositAccount
        // ... send money out ...
    }
}
```

Why this violates LSP:

- The base type Account implies withdraw is supported when conditions are met.
- FixedDepositAccount**breaks** that promise by throwing UnsupportedOperationException for all withdrawals.
- Clients coded against Account will break when given a FixedDepositAccount.

✅ **LSP-Compliant Designs**

```
import java.math.BigDecimal;
import java.time.LocalDate;
interface Account {
    BigDecimal getBalance();
}
interface Depositable {
    void deposit(BigDecimal amount);
}
interface Withdrawable {
    WithdrawalResult withdraw(BigDecimal amount, LocalDate today);
}
enum WithdrawalStatus {
    SUCCESS, INSUFFICIENT_FUNDS, LIMIT_EXCEEDED, NOT_ALLOWED
}
final class WithdrawalResult {
```

```
    private final WithdrawalStatus status;
    private final BigDecimal previousBalance;
    private final BigDecimal newBalance;
    private final String message;
    private WithdrawalResult(WithdrawalStatus status, BigDecimal pre
v, BigDecimal next, String msg) {
        this.status = status;
        this.previousBalance = prev;
        this.newBalance = next;
        this.message = msg;
    }
    public static WithdrawalResult success(BigDecimal prev, BigDecim
al next) {
        return new WithdrawalResult(WithdrawalStatus.SUCCESS, prev,
next, "OK");
    }
    public static WithdrawalResult fail(WithdrawalStatus status, Big
Decimal current, String msg) {
        return new WithdrawalResult(status, current, current, msg);
    }
    public WithdrawalStatus getStatus() { return status; }
    public BigDecimal getPreviousBalance() { return previousBalance;
 }
    public BigDecimal getNewBalance() { return newBalance; }
    public String getMessage() { return message; }
}
class SavingsAccount implements Account, Depositable, Withdrawable {
    private BigDecimal balance = BigDecimal.ZERO;
    private final BigDecimal perTxnLimit;
    public SavingsAccount(BigDecimal perTxnLimit) {
```

```
        this.perTxnLimit = perTxnLimit;
    }
    @Override
    public BigDecimal getBalance() { return balance; }
    @Override
    public void deposit(BigDecimal amount) {
        if (amount.signum() <= 0) throw new IllegalArgumentException
("Amount must be positive");
        balance = balance.add(amount);
    }
    @Override
    public WithdrawalResult withdraw(BigDecimal amount, LocalDate to
day) {
        if (amount.signum() <= 0) return WithdrawalResult.fail(
                WithdrawalStatus.NOT_ALLOWED, balance, "Amount must
be positive");
        if (amount.compareTo(perTxnLimit) > 0)
            return WithdrawalResult.fail(WithdrawalStatus.LIMIT_EXCE
EDED, balance, "Over per-transaction limit");
        if (balance.compareTo(amount) < 0)
            return WithdrawalResult.fail(WithdrawalStatus.INSUFFICIE
NT_FUNDS, balance, "Insufficient funds");
        BigDecimal prev = balance;
        balance = balance.subtract(amount);
        return WithdrawalResult.success(prev, balance);
    }
}
class CurrentAccount implements Account, Depositable, Withdrawable {
    private BigDecimal balance = BigDecimal.ZERO;
    private final BigDecimal overdraftLimit; // e.g., 50,000
```

```
    public CurrentAccount(BigDecimal overdraftLimit) {
        this.overdraftLimit = overdraftLimit.abs();
    }
    @Override
    public BigDecimal getBalance() { return balance; }
    @Override
    public void deposit(BigDecimal amount) {
        if (amount.signum() <= 0) throw new IllegalArgumentException
("Amount must be positive");
        balance = balance.add(amount);
    }
    @Override
    public WithdrawalResult withdraw(BigDecimal amount, LocalDate to
day) {
        if (amount.signum() <= 0) return WithdrawalResult.fail(
                WithdrawalStatus.NOT_ALLOWED, balance, "Amount must
be positive");
        BigDecimal newBalance = balance.subtract(amount);
        if (newBalance.compareTo(overdraftLimit.negate()) < 0) {
            return WithdrawalResult.fail(WithdrawalStatus.INSUFFICIE
NT_FUNDS, balance,
                    "Would exceed overdraft limit");
        }
        BigDecimal prev = balance;
        balance = newBalance;
        return WithdrawalResult.success(prev, balance);
    }
}
class FixedDepositAccount implements Account, Depositable {
    private BigDecimal balance = BigDecimal.ZERO;
```

```
    @Override
    public BigDecimal getBalance() { return balance; }
    @Override
    public void deposit(BigDecimal amount) {
        if (amount.signum() <= 0) throw new IllegalArgumentException
("Amount must be positive");
        balance = balance.add(amount);
    }
    // No withdraw method: this account simply is NOT Withdrawable
}
// Client code that respects capabilities:
class PaymentService {
    public WithdrawalResult payOut(Withdrawable source, BigDecimal a
mount) {
        return source.withdraw(amount, LocalDate.now()); // Works fo
r SavingsAccount and CurrentAccount
    }
}
```

Guidelines for LSP:-

1. Signature Rule:
2. Property Rule
3. Method Rule

✅ **1. Signature Rule**

Definition:
A subtype must maintain the **method signature compatibility** of its supertype.

Subtypes of Signature Rule

- **Parameter Rule:**
    - Subclass cannot change the parameter list of an overridden method.
    - Overloading is allowed, but it does not count as overriding.
- **Return Type Rule:**

    - Subclass can return the same type or a **covariant type** (a subtype of the original return type).
- **Exception Rule:**
    - Subclass cannot throw **broader checked exceptions** than the base class.
- **Access Modifier Rule:**
    - Subclass cannot make the method **less accessible** (e.g., cannot change public to protected).

✅ 2. Property Rule

Definition:
A subtype must preserve the **invariants (properties)** of the base class.

Subtypes of Property Rule

- **State Invariant Rule:**
    - Subclass must maintain the same constraints on object state as the base class.
    - Example: If balance >= 0 in Account, subclass cannot allow negative balance unless explicitly allowed.
- **Representation Invariant Rule:**
    - Internal representation consistency must be preserved.
    - Example: If base class ensures interestRate is always positive, subclass cannot allow negative interest.

✅ 3. Method Rule

Definition:
A subtype must not **strengthen preconditions** or **weaken postconditions** of the base class methods.

Subtypes of Method Rule

- **Precondition Rule:**
    - Subclass cannot require **more** than the base class.
    - Example: Base allows withdraw(amount > 0), subclass cannot require amount > 100.
- **Postcondition Rule:**

    - Subclass cannot guarantee **less** than the base class.
    - Example: If base guarantees balance decreases by amount, subclass cannot skip updating balance.
- **Exception Behavior Rule:**
    - Subclass cannot introduce new failure modes that base class didn't allow (e.g., throwing UnsupportedOperationException unexpectedly).

✅ **Summary Table**

Rule	Subtypes	Example Violation
Signature Rule	Parameter, Return Type, Exception, Access Modifier	Changing method parameters or throwing broader exceptions
Property Rule	State Invariant, Representation Invariant	Allowing negative balance when base forbids it
Method Rule	Precondition, Postcondition, Exception Behavior	Adding stricter withdrawal conditions

Signature = Method Name+ Method Argument + Method Return Type

Interface Segregation Principle (ISP)

This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that **"do not force any client to implement an interface which is irrelevant to them"**. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

Example:

```
//ISP Voilated
interface Shape{
   void area();
   void volume();
}
```

```
class Rectangle implements Shape{
    int height=2,width=3;
    public void area(){
        System.out.println("Area of Rectange = "+ height*width);
    }
    public void volume(){
        throw new IllegalArgumentException("This operation is not
allowed in Rectangle.");
    }
}

class Cube implements Shape{
    int side=4;
    public void area(){
        System.out.println("Area of Cube="+ 6*side*side);
    }
    public void volume(){
        System.out.println("Volume of Cube="+ side*side*side);
    }
}
class Main {
    public static void main(String[] args) {
        Shape rect= new Rectangle();
        rect.area();
        // rect.volume();

        Shape cube=new Cube();
        cube.area();
        cube.volume();
```

```
    }
}
```

//ISP Followed

```
interface TwoDimestionShape{
   void area();
}

interface ThreeDimestionShape{
   void area();
   void volume();
}

class Rectangle implements TwoDimestionShape{
    int height=2,width=3;
    public void area(){
        System.out.println("Area of Rectange = "+ height*width);
    }
}

class Cube implements ThreeDimestionShape{
    int side=4;
    public void area(){
        System.out.println("Area of Cube="+ 6*side*side);
    }
    public void volume(){
        System.out.println("Volume of Cube="+ side*side*side);
    }
}
```

```
class Main {
    public static void main(String[] args) {
        TwoDimestionShape rect= new Rectangle();
        rect.area();

        ThreeDimestionShape cube=new Cube();
        cube.area();
        cube.volume();
    }
}

//
```

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) is a principle in object-oriented design that states that "**High-level modules should not depend on low-level modules. Both should depend on abstractions**". Additionally, abstractions should not depend on details. Details should depend on abstractions.

- In simpler terms, the DIP suggests that classes should rely on abstractions (e.g., interfaces or abstract classes) rather than concrete implementations.
- This allows for more flexible and decoupled code, making it easier to change implementations without affecting other parts of the codebase.

Example: In a software development team, developers depend on an abstract version control system (e.g., Git) to manage and track changes to the codebase. They don't depend on specific details of how Git works internally.

- This allows developers to focus on writing code without needing to understand the intricacies of version control implementation. Below is the code of above example to understand Dependency Inversion Principle:

```
// Interface for version control system
interface IVersionControl {
```

```
    void commit(String message);
    void push();
    void pull();
}

// Git version control implementation
class GitVersionControl implements IVersionControl {
    @Override
    public void commit(String message) {
        System.out.println("Committing changes to Git with message:
" + message);
    }

    @Override
    public void push() {
        System.out.println("Pushing changes to remote Git
repository.");
    }

    @Override
    public void pull() {
        System.out.println("Pulling changes from remote Git
repository.");
    }
}

// Team class that relies on version control
class DevelopmentTeam {
    private IVersionControl versionControl;
```

```
    public DevelopmentTeam(IVersionControl vc) {
        this.versionControl = vc;
    }

    public void makeCommit(String message) {
        versionControl.commit(message);
    }

    public void performPush() {
        versionControl.push();
    }

    public void performPull() {
        versionControl.pull();
    }
}

public class Main {
    public static void main(String[] args) {
        GitVersionControl git = new GitVersionControl();
        DevelopmentTeam team = new DevelopmentTeam(git);

        team.makeCommit("Initial commit");
        team.performPush();
        team.performPull();
    }
}
```

Need for SOLID Principles in Object-Oriented Design

Below are some of the main reasons why solid principles are important in object oriented design:

- SOLID principles make code easier to maintain. When each class has a clear responsibility, it's simpler to find where to make changes without affecting unrelated parts of the code.
- These principles support growth in software. For example, the Open/Closed Principle allows developers to add new features without changing existing code, making it easier to adapt to new requirements.
- SOLID encourages flexibility. By depending on abstractions rather than specific implementations (as in the Dependency Inversion Principle), developers can change components without disrupting the entire system

What is the Singleton pattern? How do you implement it in Java?

Method Hiding

If a subclass declares a static method with the **same signature** as a static method in its superclass, the subclass method **hides** (it does not override) the superclass method. This mechanism happens because the static method is resolved at the compile time. Static method bind during the compile time using the type of reference not a type of object.

Examples:

```
// Base Class
class Complex {
    public static void f1()
    {
        System.out.println(
            "f1 method of the Complex class is executed.");
    }
}

// class child extend Demo class
class Sample extends Complex {
    public static void f1()
```

```
    {
        System.out.println(
            "f1 of the Sample class is executed.");
    }
}
public class Main {

    public static void main(String args[])
    {
        Complex d1 = new Complex();

        // d2 is reference variable of class Demo that
        // points to object of class Sample
        Complex d2 = new Sample();

        // But here method will be call using type of reference
        d1.f1();
        d2.f1();
    }
}
```

Output

f1 method of the Complex class is executed.

f1 method of the Complex class is executed.

Program Demonstrate the Difference Between Method Overriding and Method Hiding in Java:

```
class Complex {
    public static void f1()
    {
```

```
        System.out.println("f1 method of the Complex class is
executed.");
    }

    public void f2()
    {
        System.out.println("f2 method of the Complex class is
executed.");
    }
}

class Sample extends Complex {
    public static void f1()
    {
        System.out.println("f1 of the Sample class is executed.");
    }
    public void f2()
    {
        System.out.println("f2 method of the Sample class is
executed.");
    }
}
public class Main {
    public static void main(String args[])
    {
        Complex d1 = new Complex();

         // d2 is the reference variable of class Demo that
        // points to object of class Sample
        Complex d2 = new Sample();
```

```
        // But here method will be called using type of
        // reference
        d1.f1();
        d2.f1();

        // ***** Function overriding in java *****//
        // method refer to the  object  in the method
        // overriding.
        d1.f2();
        d2.f2();
    }
}
```

Output

f1 method of the Complex class is executed.

f1 method of the Complex class is executed.

f2 method of the Complex class is executed.

f2 method of the Sample class is executed.

Difference Between Method Overriding and Method Hiding in Java

- In method overriding both the method parent class and child class are non-static.
- In method Hiding both the method parent class and child class are static.
- In method Overriding method resolution is done on the basis of the Object type.
- In method Hiding method resolution is done on the basis of reference type.
- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.
- In method overriding, we have the ability to use the "super" keyword to explicitly access superclass methods (super keyword is non-static reference variable we

cant use it in static method as we know static methods can only access static members of the class). Therefore, method overriding does not involve method hiding since the "super" keyword allows us to access and invoke superclass methods when needed.

✅ What is Multitasking?

Multitasking means performing **multiple tasks at the same time**. In an operating system, this usually refers to running multiple programs or processes concurrently.

- **Example**: Listening to music while browsing the web and downloading a file.
- **Types of multitasking**:
    1. **Process-based multitasking**: Multiple processes run simultaneously (e.g., Word + Chrome + Spotify).
    2. **Thread-based multitasking**: Multiple threads within the same process run concurrently (e.g., a browser loading multiple tabs).

✅ What is Multithreading?

Multithreading is a type of multitasking where a **single process** is divided into multiple **threads**, and these threads run concurrently.

- Each thread is a lightweight sub-process.
- Threads share the same memory space of the process but execute independently.
- **Example**: A text editor:
    - One thread handles typing input.
    - Another thread handles spell-check.
    - Another thread handles autosave.

✅ How are Multitasking and Multithreading Related?

- Multithreading is a subset of multitasking.
- Multitasking can be:
    - **Process-based** (multiple programs)
    - **Thread-based** (multiple threads in one program)

- Multithreading improves **responsiveness** and **resource utilization** within a single application.

Quick Analogy

- **Multitasking** = A chef cooking multiple dishes at once (different stoves = different processes).
- **Multithreading** = For one dish, the chef chops vegetables while the assistant stirs the sauce (different threads in the same process)

CPU

The CPU, often referred to as the brain of the computer, is responsible for executing instructions from programs. It performs basic arithmetic, logic, control, and input/output operations specified by the instructions.

Core

A core is an individual processing unit within a CPU. Modern CPUs can have multiple cores, allowing them to perform multiple tasks simultaneously.

A quad-core processor has four cores, allowing it to perform four tasks simultaneously. For instance, one core could handle your web browser, another your music player, another a download manager, and another a background system update.

Program

A program is a set of instructions written in a programming language that tells the computer how to perform a specific task.

Microsoft Word is a program that allows users to create and edit documents.

Process

A process is a running instance of a program. When a program runs, the operating system creates a process to manage its execution.

When we open Microsoft Word, it becomes a process in the operating system.

Thread

A thread is the smallest unit of execution within a process. A process can have multiple threads, which share the same resources but can run independently.

A web browser like Chrome uses **multiple processes** (one per tab or site instance) and **multiple threads within each process** for tasks like rendering, networking, and JavaScript execution.

Multitasking

Multitasking allows an operating system to run multiple processes simultaneously. On single-core CPUs, this is done through time-sharing, rapidly switching between tasks. On multi-core CPUs, true parallel execution occurs, with tasks distributed across cores. The OS scheduler balances the load, ensuring efficient and responsive system performance.

Example: We are browsing the internet while listening to music and downloading a file.

Multitasking utilizes the capabilities of a CPU and its cores. When an operating system performs multitasking, it can assign different tasks to different cores. This is more efficient than assigning all tasks to a single core.

Multithreading

Multithreading refers to the ability to execute multiple threads within a single process concurrently.

A web browser can use multithreading by having separate threads for rendering the page, running JavaScript, and managing user inputs. This makes the browser more responsive and efficient.

Multithreading enhances the efficiency of multitasking by breaking down individual tasks into smaller sub-tasks or threads. These threads can be processed simultaneously, making better use of the CPU's capabilities.

In a single-core system:

Both threads and processes are managed by the OS scheduler through time slicing and context switching to create the illusion of simultaneous execution.

In a multi-core system:

Both threads and processes can run in true parallel on different cores, with the OS scheduler distributing tasks across the cores to optimize performance.

Time Slicing

- **Definition:** Time slicing divides CPU time into small intervals called time slices or quanta.
- **Function:** The OS scheduler allocates these time slices to different processes and threads, ensuring each gets a fair share of CPU time.
- **Purpose:** This prevents any single process or thread from monopolizing the CPU, improving responsiveness and enabling concurrent execution.

Context Switching

- **Definition:** Context switching is the process of saving the state of a currently running process or thread and loading the state of the next one to be executed.
- **Function:** When a process or thread's time slice expires, the OS scheduler performs a context switch to move the CPU's focus to another process or thread.
- **Purpose:** This allows multiple processes and threads to share the CPU, giving the appearance of simultaneous execution on a single-core CPU or improving parallelism on multi-core CPUs.

Multitasking can be achieved through multithreading where each task is divided into threads that are managed concurrently.

While multitasking typically refers to the running of multiple applications, multithreading is more granular, dealing with multiple threads within the same application or process.

Example:

I have two applications open: Word and a web browser. Both are running at the same time, so this is multitasking. For each application, a separate process is created. Suppose I am typing something in Word and spell checking is happening simultaneously—these two are threads.

Multithreading in Java

Java provides robust support for multithreading, allowing developers to create applications that can perform multiple tasks simultaneously, improving performance and responsiveness.

In Java, multithreading is the concurrent execution of two or more threads to maximize the utilization of the CPU. Java's multithreading capabilities are part of the java.lang package, making it easy to implement concurrent execution.

In a single-core environment, Java's multithreading is managed by the JVM and the OS, which switch between threads to give the illusion of concurrency.

The threads share the single core, and time-slicing is used to manage thread execution.

In a multi-core environment, Java's multithreading can take full advantage of the available cores.

The JVM can distribute threads across multiple cores, allowing true parallel execution of threads.

A thread is a lightweight process, the smallest unit of processing. Java supports multithreading through its java.lang.Thread class and the java.lang.Runnable interface.

When a Java program starts, one thread begins running immediately, which is called the main thread. This thread is responsible for executing the main method of a program.

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Hello world !");
    }
}
```

To create a new thread in Java, you can either extend the Thread class or implement the Runnable interface.

Method 1: extend the Thread class

1. A new class World is created that extends Thread.
2. The run method is overridden to define the code that constitutes the new thread.
3. start method is called to initiate the new thread.

```
public class Test {
    public static void main(String[] args) {
        World world = new World();
        world.start();
        for (; ; ) {
            System.out.println("Hello");
        }
    }
}
```

```
public class World extends Thread {
    @Override
    public void run() {
        for (; ; ) {
            System.out.println("World");
        }
    }
}
```

Method 2: Implement Runnable interface

1. A new class World is created that implements Runnable.
2. The run method is overridden to define the code that constitutes the new thread.
3. A Thread object is created by passing an instance of World.
4. start method is called on the Thread object to initiate the new thread.

```
public class Test {
    public static void main(String[] args) {
        World world = new World();
        Thread thread = new Thread(world);
        thread.start();
        for (; ; ) {
            System.out.println("Hello");
        }
    }
}
```

```
public class World implements Runnable {
    @Override
    public void run() {
        for (; ; ) {
            System.out.println("World");
        }
    }
}
```

Thread Lifecycle

The lifecycle of a thread in Java consists of several states, which a thread can move through during its execution.

- **New**: A thread is in this state when it is created but not yet started.
- **Runnable**: After the start method is called, the thread becomes runnable. It's ready to run and is waiting for CPU time.
- **Running**: The thread is in this state when it is executing.
- **Blocked/Waiting**: A thread is in this state when it is waiting for a resource or for another thread to perform an action.
- **Terminated**: A thread is in this state when it has finished executing.

```
public class MyThread extends Thread{
    @Override
    public void run() {
        System.out.println("RUNNING"); // RUNNING
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
```

```
    }

    public static void main(String[] args) throws
InterruptedException {
        MyThread t1 = new MyThread();
        System.out.println(t1.getState()); // NEW
        t1.start();
        System.out.println(t1.getState()); // RUNNABLE
        Thread.sleep(100);
        System.out.println(t1.getState()); // TIMED_WAITING
        t1.join();
        System.out.println(t1.getState()); // TERMINATED
    }
}
```

Runnable vs Thread

Use Runnable when you want to separate the task from the thread, allowing the class to extend another class if needed. Extend Thread if you need to override Thread methods or if the task inherently requires direct control over the thread itself, though this limits inheritance.

Thread methods

1. **start( ):** Begins the execution of the thread. The Java Virtual Machine (JVM) calls the run() method of the thread.
2. **run( ):** The entry point for the thread. When the thread is started, the run() method is invoked. If the thread was created using a class that implements Runnable, the run() method will execute the run() method of that Runnable object.
3. **sleep(long millis):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **join( ):** Thread.join() lets one thread **wait** for another thread to **finish** before it continues. It's a way to **coordinate** thread completion.

   Typical use case: Start multiple worker threads, then wait for them to finish before proceeding (e.g., aggregating results).

   ```
   public class MyThread{

       public static void main(String[] args) {

          Thread1 t1=new Thread1();

          Thread2 t2=new Thread2();
   ```

```
        System.out.println("Thread Extecution Started");
        t1.start();
        t2.start();
        try {
             t1.join();
             t2.join();
         } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
         }
        System.out.println("Thread Extecution Completed1");
        System.out.println("Thread Extecution Completed2");
    }
}
class Thread1 extends Thread{
     @Override
    public void run() {
        for(int i=0;i<10;i++)
            System.out.println("Thread1");
    }
}
class Thread2 extends Thread{
     @Override
    public void run() {
        for(int i=0;i<10;i++)
            System.out.println("Thread2");
    }
}
```

5. **setPriority(int newPriority):** Changes the priority of the thread. The priority is a value between Thread.MIN_PRIORITY (1) and Thread.MAX_PRIORITY (10).

```
public class MyThread extends Thread {
    public MyThread(String name) {  //Setting the thread name
        super(name);
    }

    @Override
    public void run() {
        System.out.println("Thread is Running...");
        for (int i = 1; i <= 5; i++) {
            for (int j = 0; j < 5; j++) {
                System.out.println(Thread.currentThread().getName()
+ " - Priority: " + Thread.currentThread().getPriority() + " -
count: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();

                }
            }
        }
    }

    public static void main(String[] args) throws
InterruptedException {

        MyThread l = new MyThread("Low Priority Thread");
        MyThread m = new MyThread("Medium Priority Thread");
        MyThread n = new MyThread("High Priority Thread");
        l.setPriority(Thread.MIN_PRIORITY);
        m.setPriority(Thread.NORM_PRIORITY);
        n.setPriority(Thread.MAX_PRIORITY);
        l.start();
        m.start();
        n.start();

    }
}
```

6. interrupt(): Interrupts the thread. If the thread is blocked in a call to wait(), sleep(), or join(), it will throw an InterruptedException**.**

When we say interrupts the thread, it does NOT mean the thread is forcefully stopped or killed. It simply means: Java sends a request (a signal) to the thread asking it to stop what it is doing. This request is done by setting the interrupt flag of the thread to true.

7. yield(): Thread.yield() is a static method that suggests the current thread temporarily pause its execution to allow other threads of the same or higher priority to execute. It's important to note that yield() is just a hint to the thread scheduler, and the actual behavior may vary depending on the JVM and OS.

```
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + "
is running...");
            Thread.yield();
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.start();
        t2.start();
    }
}
```

8. Thread.setDaemon(boolean): Marks the thread as either a daemon thread or a user thread. When the JVM exits, all daemon threads are terminated. Daemon threads are those threads which runs in background. JVM doesn't wait for daemon threads. JVM only wait for user thread or main thread.

```
public class MyThread extends Thread {
    @Override
    public void run() {
        while (true) {
            System.out.println("Hello world! ");
        }
    }

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.setDaemon(true); // myThread is daemon thread (
like Garbage collector ) now
```

```
        MyThread t1 = new MyThread();
        t1.start(); // t1 is user thread
        myThread.start();
        System.out.println("Main Done");
    }
}
```

Synchronisation

Let's see an example where two threads are incrementing same couter.

```
class Counter {
    private int count = 0; // shared resource

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class MyThread extends Thread {
    private Counter counter;

    public MyThread(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }

    public static void main(String[] args) {
        Counter counter = new Counter();
        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);
        t1.start();
        t2.start();
        try {
```

```
            t1.join();
            t2.join();
        }catch (Exception e){

        }
        System.out.println(counter.getCount()); // Expected: 2000,
Actual will be random <= 2000
    }
}
```

The output of the code is not 2000 because the increment method in the Counter class is not synchronized. This results in a race condition when both threads try to increment the count variable concurrently.

Without synchronization, one thread might read the value of count before the other thread has finished writing its incremented value. This can lead to both threads reading the same value, incrementing it, and writing it back, effectively losing one of the increments.

We can fix this by using synchronized keyword

```
class Counter {
    private int count = 0; // shared resource

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

OR

```
class Counter {

    private int count = 0; // shared resource


    public void increment() {

        synchronized (this) {   // Locks on the current Counter
instance

            count++;
```

```
        }
    }


    public int getCount() {
         return count;
    }
}
```

By synchronizing the increment method, you ensure that only one thread can execute this method at a time, which prevents the race condition. With this change, the output will consistently be 2000.

Critical Section

In **multithreading**, a **critical section** refers to a part of the code that **accesses shared resources** (like variables, data structures, or files) and **must not be executed by more than one thread at the same time** to avoid data inconsistency or race conditions.

Mutual Exclusion

Mutual Exclusion means **only one thread or process can access a shared resource at a time**.

✅ **Why do we need it?**

- In multithreading, multiple threads may try to **read and write the same data** simultaneously.
- Without control, this leads to **race conditions** (inconsistent or wrong results).

There are 2 types of Lock

1. **Intrinsic-** These are built into every object in Java. We don't see them, but they are there. When we use a synchronized keyword, we are using these automatic locks.
2. **Explicit-** These are more advanced locks you can control yourself using the Lock class from java.util.concurrent.locks. We explicitly say when to lock and unlock, giving you more control over how and when people can write in the notebook.

Locks

The synchronized keyword in Java provides basic thread-safety but has limitations: it locks the entire method or block, leading to potential performance issues. It lacks a try-

lock mechanism, causing threads to block indefinitely, increasing the risk of deadlocks. Additionally, synchronized doesn't support multiple condition variables, offering only a single monitor per object with basic wait/notify mechanisms. In contrast, explicit locks (Lock interface) offer finer-grained control, try-lock capabilities to avoid blocking, and more sophisticated thread coordination through multiple condition variables, making them more flexible and powerful for complex concurrency scenarios.

Lock is an interface ReentrantLock is an implementation class of Lock.

tryLock() — Non-blocking, immediate

Attempts to acquire the lock **immediately**.

- If the lock is free → acquires and returns true.
- If the lock is held by another thread → returns false **right away**.

tryLock(long time, TimeUnit unit) — Timed, interruptible wait

Attempts to acquire the lock, **waiting up to the given time**.

- If acquired within the timeout → returns true.
- If timeout elapses without acquiring → returns false.

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BankAccount {
    private int balance = 100;
    private final Lock lock = new ReentrantLock();

    public void withdraw(int amount) {
        System.out.println(Thread.currentThread().getName() + "
attempting to withdraw " + amount);
        try {
            if (lock.tryLock(1000, TimeUgunit.MILLISECONDS)) {
                if (balance >= amount) {
                    try {

System.out.println(Thread.currentThread().getName() + " proceeding
with withdrawal");
                        Thread.sleep(3000); // Simulate time taken
```

```
to process the withdrawal
                        balance -= amount;

System.out.println(Thread.currentThread().getName() + " completed
withdrawal. Remaining balance: " + balance);
                    } catch (Exception e) {
                        Thread.currentThread().interrupt();
                    } finally {
                        lock.unlock();
                    }
                } else {

System.out.println(Thread.currentThread().getName() + " insufficient
balance");
                }
            } else {
                System.out.println(Thread.currentThread().getName()
+ " could not acquire the lock, will try later");
            }
        } catch (Exception e) {
            Thread.currentThread().interrupt();
        }
    }
}
public class Main {
    public static void main(String[] args) {
        BankAccount sbi = new BankAccount();
        Runnable task = new Runnable() {
            @Override
            public void run() {
                sbi.withdraw(50);
            }
        };
        Thread t1 = new Thread(task, "Thread 1");
        Thread t2 = new Thread(task, "Thread 2");
        t1.start();
        t2.start();
    }
}
```

Reentrant Lock

A Reentrant Lock in Java is a type of lock that allows a thread to acquire the same lock multiple times without causing a deadlock. If a thread already holds the lock, it can re-enter the lock without being blocked. This is useful when a thread needs to repeatedly enter synchronized blocks or methods within the same execution flow. The ReentrantLock class from the java.util.concurrent.locks package provides this functionality, offering more flexibility than the synchronized keyword, including try-locking, timed locking, and multiple condition variables for advanced thread coordination.

```
public class ReentrantExample {
    private final Lock lock = new ReentrantLock();

    public void outerMethod() {
        lock.lock();
        try {
            System.out.println("Outer method");
            innerMethod();
        } finally {
            lock.unlock();
        }
    }

    public void innerMethod() {
        lock.lock();
        try {
            System.out.println("Inner method");
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        ReentrantExample example = new ReentrantExample();
        example.outerMethod();
    }
}
```

Methods of ReentrantLock

lock()

- Acquires the lock, blocking the current thread until the lock is available. It would block the thread until the lock becomes available, potentially leading to situations where a thread waits indefinitely.

- If the lock is already held by another thread, the current thread will wait until it can acquire the lock.

tryLock()

- Tries to acquire the lock without waiting. Returns true if the lock was acquired, false otherwise.
- This is non-blocking, meaning the thread will not wait if the lock is not available.

tryLock(long timeout, TimeUnit unit)

- Attempts to acquire the lock, but with a timeout. If the lock is not available, the thread waits for the specified time before giving up. It is used when you want to attempt to acquire the lock without waiting indefinitely. It allows the thread to proceed with other work if the lock isn't available within the specified time. This approach is useful to avoid deadlock scenarios and when you don't want a thread to block forever waiting for a lock.
- Returns true if the lock was acquired within the timeout, false otherwise.

unlock()

- Releases the lock held by the current thread.
- Must be called in a finally block to ensure that the lock is always released even if an exception occurs.

lockInterruptibly()

- Acquires the lock unless the current thread is interrupted. This is useful when you want to handle interruptions while acquiring a lock.

Read Write Lock

A Read-Write Lock is a concurrency control mechanism that allows multiple threads to read shared data simultaneously while restricting write access to a single thread at a time. This lock type, provided by the ReentrantReadWriteLock class in Java, optimizes performance in scenarios with frequent read operations and infrequent writes. Multiple readers can acquire the read lock without blocking each other, but when a thread needs to write, it must acquire the write lock, ensuring exclusive access. This prevents data inconsistency while improving read efficiency compared to traditional locks, which block all access during write operations.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
```

```
public class ReadWriteCounter {
    private int count = 0;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();

    public void increment() {
        writeLock.lock();
        try {
            count++;
            Thread.sleep(50);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        } finally {
            writeLock.unlock();
        }
    }

    public int getCount() {
        readLock.lock();
        try {
            return count;
        } finally {
            readLock.unlock();
        }
    }

    public static void main(String[] args) throws
InterruptedException {
        ReadWriteCounter counter = new ReadWriteCounter();

        Runnable readTask = new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {

System.out.println(Thread.currentThread().getName() + " read: " +
counter.getCount());
                }
            }
        };

        Runnable writeTask = new Runnable() {
```

```
            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    counter.increment();

System.out.println(Thread.currentThread().getName() + "
incremented");
                }
            }
        };

        Thread writerThread = new Thread(writeTask);
        Thread readerThread1 = new Thread(readTask);
        Thread readerThread2 = new Thread(readTask);

        writerThread.start();
        readerThread1.start();
        readerThread2.start();

        writerThread.join();
        readerThread1.join();
        readerThread2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Fairness of Locks

Fairness in the context of locks refers to the order in which threads acquire a lock. A fair lock ensures that threads acquire the lock in the order they requested it, preventing thread starvation. With a fair lock, if multiple threads are waiting, the longest-waiting thread is granted the lock next. However, fairness can lead to lower throughput due to the overhead of maintaining the order. Non-fair locks, in contrast, allow threads to “cut in line,” potentially offering better performance but at the risk of some threads waiting indefinitely if others frequently acquire the lock.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class FairnessLockExample {
    private final Lock lock = new ReentrantLock(true);
```

```
    public void accessResource() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + "
acquired the lock.");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            System.out.println(Thread.currentThread().getName() + "
released the lock.");
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        FairnessLockExample example = new FairnessLockExample();

        Runnable task = new Runnable() {
            @Override
            public void run() {
                example.accessResource();
            }
        };

        Thread thread1 = new Thread(task, "Thread 1");
        Thread thread2 = new Thread(task, "Thread 2");
        Thread thread3 = new Thread(task, "Thread 3");

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

Deadlock

A deadlock occurs in concurrent programming when two or more threads are blocked forever, each waiting for the other to release a resource. This typically happens when threads hold locks on resources and request additional locks held by other threads. For example, Thread A holds Lock 1 and waits for Lock 2, while Thread B holds Lock 2 and waits for Lock 1. Since neither thread can proceed, they remain stuck in a deadlock

state. Deadlocks can severely impact system performance and are challenging to debug and resolve in multi-threaded applications.

Deadlock typically occurs when 4 conditions are met simultaneously:

1. Mutual Exclusion: - Only one thread can access a resource at a time.
2. Hold & Wait: - A thread holding at least one resource is waiting to acquire additional resources held by other threads.
3. No preemption: - Resources cannot be forcefully taken from threads holding them.
4. Circular Wait: - A set of threads is waiting for each other in a circular chain.

```
class Pen {
    public synchronized void writeWithPenAndPaper(Paper paper) {
        System.out.println(Thread.currentThread().getName() + " is
using pen " + this + " and trying to use paper " + paper);
        paper.finishWriting();
    }

    public synchronized void finishWriting() {
        System.out.println(Thread.currentThread().getName() + "
finished using pen " + this);
    }
}

class Paper {
    public synchronized void writeWithPaperAndPen(Pen pen) {
        System.out.println(Thread.currentThread().getName() + " is
using paper " + this + " and trying to use pen " + pen);
        pen.finishWriting();
    }

    public synchronized void finishWriting() {
        System.out.println(Thread.currentThread().getName() + "
finished using paper " + this);
    }
}

class Task1 implements Runnable {
    private Pen pen;
    private Paper paper;

    public Task1(Pen pen, Paper paper) {
```

```
        this.pen = pen;
        this.paper = paper;
    }

    @Override
    public void run() {
        pen.writeWithPenAndPaper(paper); // thread1 locks pen and
tries to lock paper
    }
}

class Task2 implements Runnable {
    private Pen pen;
    private Paper paper;

    public Task2(Pen pen, Paper paper) {
        this.pen = pen;
        this.paper = paper;
    }

    @Override
    public void run() {
        synchronized (pen){
            paper.writeWithPaperAndPen(pen); // thread2 locks paper
and tries to lock pen
        }
    }
}


public class DeadlockExample {
    public static void main(String[] args) {
        Pen pen = new Pen();
        Paper paper = new Paper();

        Thread thread1 = new Thread(new Task1(pen, paper), "Thread-
1");
        Thread thread2 = new Thread(new Task2(pen, paper), "Thread-
2");

        thread1.start();
        thread2.start();
```

```
    }
}
```

Thread communication

In a multithreading environment threads often need to communicate and coordinate with each other to accomplish a task. Without proper communication mechanisms threads might end up in inefficient busy-waiting status, leading to wastage of CPU resources and potential deadlocks.

```
class SharedResource {
    private int data;
    private boolean hasData;

    public synchronized void produce(int value) {
        while (hasData) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        data = value;
        hasData = true;
        System.out.println("Produced: " + value);
        notify();
    }

    public synchronized int consume() {
        while (!hasData){
            try{
                wait();
            }catch (InterruptedException e){
                Thread.currentThread().interrupt();
            }
        }
        hasData = false;
        System.out.println("Consumed: " + data);
        notify();
        return data;
    }
}
```

```
class Producer implements Runnable {
    private SharedResource resource;

    public Producer(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            resource.produce(i);
        }
    }
}

class Consumer implements Runnable {
    private SharedResource resource;

    public Consumer(SharedResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            int value = resource.consume();
        }
    }
}

public class ThreadCommunication {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();
        Thread producerThread = new Thread(new Producer(resource));
        Thread consumerThread = new Thread(new Consumer(resource));

        producerThread.start();
        consumerThread.start();
    }
}
```

Thread Pool

A Thread Pool is a collection of pre-created, reusable threads that are kept ready to perform tasks. Instead of creating a new thread every time you need to run something (which is costly in terms of memory and CPU), a thread pool maintains a fixed number of threads. When a task is submitted:

- If a thread is free, it immediately picks up the task and runs it.
- If all threads are busy, the task waits in a queue until a thread becomes available.
- After finishing a task, the thread does not die. It goes back to the pool and waits for the next task.

Benefits of Thread Pool

- Better Performance: Threads are reused instead of being created and destroyed repeatedly.
- Faster Response Time: Tasks don't need to wait for a new thread to be created.
- Reusability: Threads remain alive after finishing tasks and are reused for future tasks.
- Resource Management: Limits the number of concurrent threads, preventing OutOfMemoryError or CPU overload.

Thread Pool Initialization

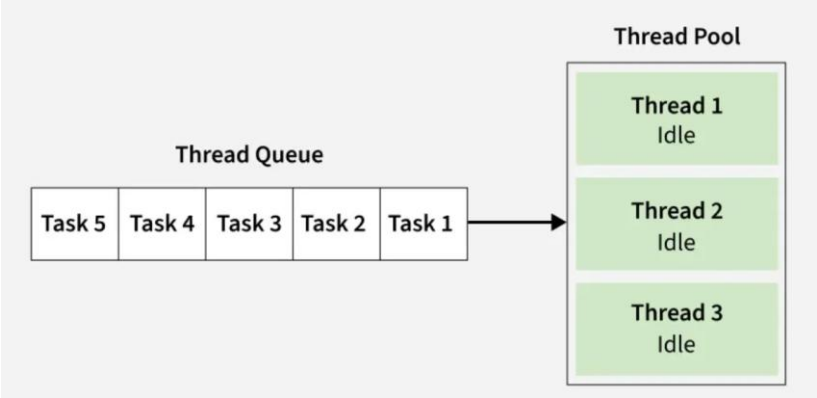
When we initialize a thread pool:

- A fixed number of worker threads are created (e.g., 3).
- These threads are kept idle, waiting for tasks.
- A task queue is set up to hold submitted tasks until a worker is free.

Thread Pool Working

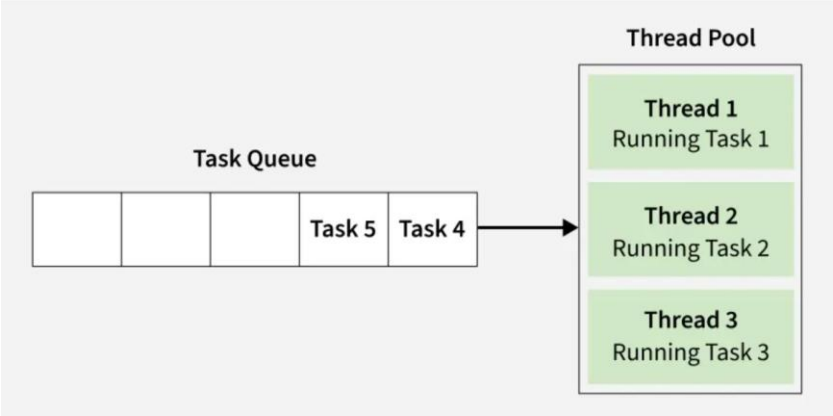
Step 1: Idle State

- Tasks are submitted and placed in the Task Queue.
- Worker threads exist but are idle until work arrives.



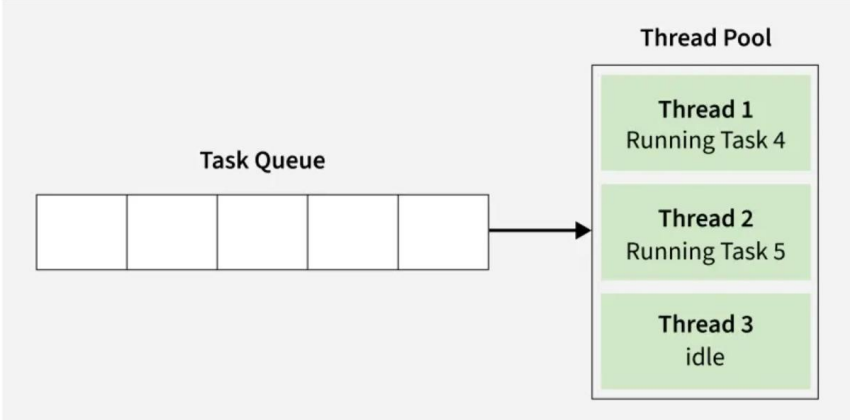
Step 2: Task Assignment

- Each idle thread picks a task from the queue.
- Example: Thread 1 -> Task 1, Thread 2 -> Task 2, Thread 3 -> Task 3
- Remaining tasks (Task 4, Task 5) wait in the queue.



Step 3: Thread Reuse

- Once a thread completes its current task, it becomes idle again.
- It immediately takes the next waiting task from the queue.
- Example: Thread 1 -> Task 4, Thread 2 -> Task 5, Thread 3 -> Idle (no tasks left)



Executors framework

The Executors framework was introduced in Java 5 as part of the java.util.concurrent package to simplify the development of concurrent applications by abstracting away many of the complexities involved in creating and managing threads.

Problems prior to Executors framework-

1. Manual Thread Management
2. Resource Management
3. Scalability
4. Thread Reuse
5. Error Handling

It will help in

1. Avoiding Manual Thread management
2. Resource management
3. Scalability
4. Thread reuse
5. Error handling

There are 3 interfaces in Executors Framework

1. Executor
2. ExecutorService
3. ScheduledExecutorService

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ExecutorFrameWork {

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 1; i < 10; i++) {
            int finalI = i;
            executor.submit(() -> {
                long result = factorial(finalI);
                System.out.println(result);
            });

        }
        executor.shutdown();
        // executor.shutdown();
        try {
```

```
                executor.awaitTermination(1, TimeUnit.SECONDS);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }

            System.out.println("Total time " +
    (System.currentTimeMillis() - startTime));
        }

        private static long factorial(int n) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            long result = 1;
            for (int i = 1; i <= n; i++) {
                result *= i;
            }
            return result;
        }
    }
```

Future

A Future interface provides methods to check if the computation is complete, to wait for its completion and to retrieve the results of the computation. The result is retrieved using Future's get() method when the computation has completed, and it blocks until it is completed.

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService =
Executors.newSingleThreadExecutor();
        Future<?> future = executorService.submit(() ->
System.out.println("Hello")); // runnable parameter
        System.out.println(future.get()); // blocking call ( null )
        if(future.isDone()){
            System.out.println("Task is done !");
        }
        executorService.shutdown();
    }
}
```

CountDownLatch

```
import java.util.concurrent.Callable;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test {
    public static void main(String[] args) throws InterruptedException {
        int n = 3;
        ExecutorService executorService = Executors.newFixedThreadPool(n);
        CountDownLatch latch = new CountDownLatch(n);
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        executorService.submit(new DependentService(latch));
        latch.await();
        System.out.println("Main");
        executorService.shutdown();
    }
}

class DependentService implements Callable<String> {
    private final CountDownLatch latch;
    public DependentService(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public String call() throws Exception {
        try {
            System.out.println(Thread.currentThread().getName() + "
service started.");
            Thread.sleep(2000);
        } finally {
            latch.countDown();
        }
        return "ok";
    }
}
```

Output:

```
pool-1-thread-3 service started.
pool-1-thread-2 service started.
pool-1-thread-1 service started.
Main
```

Cyclic Barrier

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class Main {
```

```
    public static void main(String[] args)  {
        int numberOfSubsystems = 4;
        CyclicBarrier barrier = new CyclicBarrier(numberOfSubsystems, new
Runnable() {
            @Override
            public void run() {
                System.out.println("All subsystems are up and running.
System startup complete.");
            }
        });

        Thread webServerThread = new Thread(new Subsystem("Web Server",
2000, barrier));
        Thread databaseThread = new Thread(new Subsystem("Database", 4000,
barrier));
        Thread cacheThread = new Thread(new Subsystem("Cache", 3000,
barrier));
        Thread messagingServiceThread = new Thread(new
Subsystem("Messaging Service", 3500, barrier));

        webServerThread.start();
        databaseThread.start();
        cacheThread.start();
        messagingServiceThread.start();

    }


}

class Subsystem implements Runnable {
    private String name;
    private int initializationTime;
    private CyclicBarrier barrier;

    public Subsystem(String name, int initializationTime, CyclicBarrier
barrier) {
        this.name = name;
        this.initializationTime = initializationTime;
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            System.out.println(name + " initialization started.");
            Thread.sleep(initializationTime); // Simulate time taken to
initialize
            System.out.println(name + " initialization complete.");
```

```
            barrier.await();
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

🔹 volatile Variable

A volatile variable is a **modifier** used in multithreaded programming to ensure **visibility** of changes to variables across threads.

Key Characteristics:

- Ensures that **reads and writes** to the variable are directly from and to **main memory**, not from thread-local caches.
- Guarantees **visibility**, but **not atomicity**.
- Useful for flags or simple state variables.

Example:

```
class SharedResource {
    private volatile boolean flag = false;
    public void setFlagTrue() {
        System.out.println("writerFlag made the flag true.");
        flag = true;
    }
    public void printIfFlagTrue() {
        while(!flag){
            //do nothing
        }
        System.out.println("Flag is true!");
    }
     public static void main(String[] args){
        SharedResource sharedResource = new SharedResource();
        Thread writerThread = new Thread(()->{
```

```
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){
                Thread.currentThread().interrupt();
            }
            sharedResource.setFlagTrue();
        });

        Thread readerThread = new Thread(()->{
            sharedResource.printIfFlagTrue();
        });

        writerThread.start();
        readerThread.start();
    }
}
```

In this example, if one thread sets flag to true, other threads will immediately see the updated value.

🔹 Atomic Variable

Atomic variables are part of the java.util.concurrent.atomic package and provide **atomic operations** on variables without using synchronization.

Key Characteristics:

- Ensures **atomicity**, **visibility**, and **ordering**.
- Provides methods like get(), set(), incrementAndGet(), compareAndSet(), etc.
- Ideal for counters, flags, or any variable that needs atomic updates.

Example:

```
import java.util.concurrent.atomic.AtomicInteger;
class Counter {
    private AtomicInteger count = new AtomicInteger(0);
```

```
    public void increment() {
        count.incrementAndGet();
    }
    public int getCounter() {
        return count.get();
    }
    public static void main(String[] args) throws
InterruptedException{
        Counter counter = new Counter();
        Thread thread1 = new Thread(()->{
            for(int i=0;i<1000;i++){
                counter.increment();
            }
        });

        Thread thread2 = new Thread(()->{
            for(int i=0;i<1000;i++){
                counter.increment();
            }
        });

        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println("Count="+counter.getCounter());
    }
}
```

Here, incrementAndGet() is thread-safe and atomic, so multiple threads can safely update count.

🔸 **Summary**

Feature	volatile	Atomic
Visibility	✅ Ensures visibility	✅ Ensures visibility
Atomicity	❌ Not atomic	✅ Atomic operations
Synchronization	❌ No locking	✅ Lock-free thread safety
Use Case	Flags, state indicators	Counters, accumulators, etc.

Atomicity- In Java **atomicity** refers to operations that are **indivisible**—they either happen **completely** or **not at all**, with **no intermediate state** visible to other threads.

Difference between run() and start()

In Java multithreading, start() and run() are both methods of the Thread class, but they behave very differently:

✅ 1. start() Method

- **Purpose**: Creates a new thread and calls its run() method internally.
- **Behavior**:
  - Allocates a new call stack for the thread.
  - Executes run()in a separate thread, enabling true multithreading.
- **Example**:

```
Thread t = new Thread(() -
> System.out.println("Running in: " + Thread.currentThread().getName()));
```

Output:
Running in: Thread-0 (or some other thread name)

✅ 2. run() Method

- **Purpose**: Contains the code that will run when the thread starts.

- **Behavior**:
    - If you call run()directly, it does NOT create a new thread.
    - It runs in the current thread like a normal method call.
- **Example:**

```
Thread t = new Thread(() -
> System.out.println("Running in: " + Thread.currentThread().getName()));
t.run(); // No new thread, runs in main thread
```

Output:
Running in: main

✅ Key Differences

Feature	start()	run()
Creates new thread?	Yes	No
Executes run() in	New thread	Current thread
True multithreading	Yes	No

Static Binding

The binding which can be resolved at compile time by the compiler is known as static or early binding. The binding of all the static, private, and final methods is done at compile-time.

Example:

```
class NewClass {


    // Static nested inner class
    // Class 1
    public static class superclass {


        // Method of inner class
```

```
    static void print()
    {

        // Print statement
        System.out.println(
            "print() in superclass is called");
    }
}

// Static nested inner class
// Class 2
public static class subclass extends superclass {

    // Method of inner class
    static void print()
    {

        // print statement
        System.out.println(
            "print() in subclass is called");
    }
}

// Method of main class
// Main driver method
public static void main(String[] args)
{

    // Creating objects of static inner classes
```

```
        // inside main() method
        superclass A = new superclass();
        superclass B = new subclass();

        // Calling method over above objects
        A.print();
        B.print();
    }
}
```

Output

print() in superclass is called

print() in superclass is called

Output Explanation: As you can see, in both cases the print method of the superclass is called. Let us discuss how this happens

- We have created one object of subclass and one object of the superclass with the reference of the superclass.
- Since the print method of the superclass is static, the compiler knows that it will not be overridden in subclasses and hence compiler knows during compile time which print method to call and hence no ambiguity.

Dynamic Binding

In Dynamic binding compiler doesn't decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have the same method.

Example:

```
public class GFG {

    // Static nested inner class
    // Class 1
    public static class superclass {
```

```
    // Method of inner class 1
    void print()
    {

        // Print statement
        System.out.println(
            "print in superclass is called");
    }
}

// Static nested inner class
// Class 2
public static class subclass extends superclass {

    // Method of inner class 2
    @Override void print()
    {

        // Print statement
        System.out.println(
            "print in subclass is called");
    }
}

// Method inside main class
public static void main(String[] args)
{
```

```
        // Creating object of inner class 1
        // with reference to constructor of super class
        superclass A = new superclass();

        // Creating object of inner class 1
        // with reference to constructor of sub class
        superclass B = new subclass();

        // Calling print() method over above objects
        A.print();
        B.print();
    }
}
```

Output

print in superclass is called

print in subclass is called

Output Explanation: Here the output differs. But why? Let's break down the code and understand it thoroughly.

- Methods are not static in this code.
- During compilation, the compiler has no idea as to which print has to be called since the compiler goes only by referencing variable not by the type of object, and therefore the binding would be delayed to runtime and therefore the corresponding version of the print will be called based on type on an object.

Tip: *Geeks, now the question arises why binding of static, final, and private methods is always static binding?*

Static binding is better performance-wise (no extra overhead is required). The compiler knows that all such methods ***cannot be overridden*** *and will always be accessed by objects of the local class. Hence compiler doesn't have any difficulty determining the object of the class (local class for sure). That's the reason binding for such methods is static.*

Static & Instance Initializer Block

Static and instance initializer blocks are code blocks used to initialize variables in Java. **Static blocks** run only once when the class is loaded and are used for initializing static variables. **Instance blocks** run every time an object is created and are used to initialize instance variables.

```
class Main {
    {
        System.out.println("IIB Block1");
    }
    static{
        System.out.println("Static Block");
    }
    {
        System.out.println("IIB Block2");
    }
    public static void main(String[] args) {
       Main obj1 = new Main();
       Main obj2 = new Main();
       Main obj3 = new Main();
    }
}
```

```
Static Block
IIB Block1
IIB Block2
IIB Block1
IIB Block2
IIB Block1
IIB Block2

=== Code Execu
```

Inner Class

An inner class is a class declared inside the body of another class. The inner class has access to all members (including private) of the outer class, but the outer class can access the inner class members only through an object of the inner class.

Syntax:

class OuterClass {
// Outer class members

class InnerClass {
// Inner class members
}

}

Example:

```
public class OuterClass {

  class InnerClass {

    void display() {

      System.out.println("Hello from Inner Class!");
```

```
    }
  }
  public static void main(String[] args) {
    OuterClass outer = new OuterClass();
    InnerClass inner = outer.new InnerClass();
    inner.display();
  }
}
```

Features of Inner Classes

- **Encapsulation:** Inner classes can access private members of the outer class, providing better encapsulation.
- **Code Organization:** Logically groups classes that belong together, making code more readable.
- **Access to Outer Class:** Inner class instances have a reference to the outer class instance.
- **Namespace Management:** Helps avoid naming conflicts by nesting related classes.

Types of Inner Classes

1. Member Inner Class
2. Method-Local Inner Class
3. Static Nested Class
4. Anonymous Inner Class

1. Member Inner Class

A member inner class is a non-static class defined at the member level of another class. It has access to all members of the outer class, including private members.

```
class Outer {
  private int outerVar = 100;

  // Member inner class
  class Inner {
    void display() {
      System.out.println("Outer variable: " + outerVar);
    }
  }
}
```

```
class Main {
  public static void main(String[] args) {
    // Creating inner class instance
    Outer.Inner inner = new Outer().new Inner();
    inner.display();
  }
}
```

Output: Outer variable: 100

2. Method-Local Inner Class

A method-local inner class is defined inside a method of the outer class. It can only be instantiated within the method where it is defined.

```
class Outer {
  void outerMethod() {
    System.out.println("Inside outerMethod");

    // Method-local inner class
    class Inner {
      void innerMethod() {
        System.out.println("Inside innerMethod");
      }
    }

    Inner inner = new Inner();
    inner.innerMethod();
  }
}

class Main {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.outerMethod();
  }
}
```

Output:

Inside outerMethod
Inside innerMethod

3. Static Nested Classes

A static nested class is a static class defined inside another class. It does not have access to instance members of the outer class but can access static members.

```
class Outer {
  private static int staticVar = 50;
```

```
  // Static nested class
  static class Inner {
    void display() {
      System.out.println("Static variable: " + staticVar);
    }
  }
}

class Main {
  public static void main(String[] args) {
    // No need for outer class instance
    Outer.Inner inner = new Outer.Inner();
    inner.display();
  }
}
```

Output:

Static variable: 50

4. Anonymous Inner Classes

An anonymous inner class is an inner class without a name. It is declared and instantiated in a single statement. It is used to provide a specific implementation of a class or interface on the fly.

Types of Anonymous Inner Classes:

a) As a Subclass

```
class Demo {
  void show() {
    System.out.println("Inside Demo's show method");
  }
}

class Main {
  public static void main(String[] args) {
    // Anonymous inner class extending Demo
    Demo obj = new Demo() {
      @Override
      void show() {
        super.show();
        System.out.println("Inside Anonymous class");
      }
    };

    obj.show();
  }
```

```
}
```
Output:

```
Inside Demo's show method
Inside Anonymous class
```

b) As an Interface Implementation

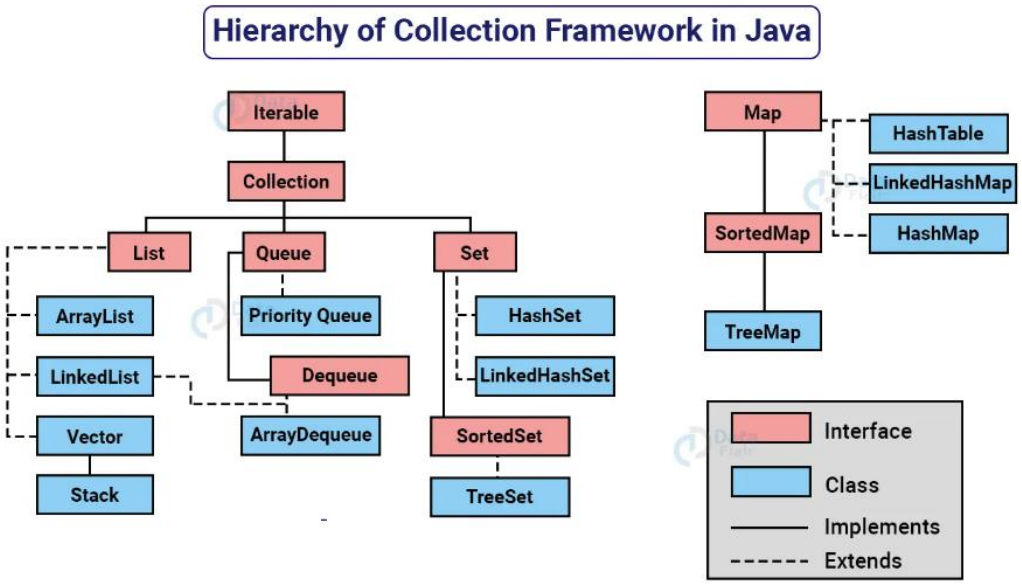
```
interface Hello {
  void greet();
}

class Main {
  public static void main(String[] args) {
    // Anonymous inner class implementing Hello
    Hello hello = new Hello() {
      @Override
      public void greet() {
        System.out.println("Hello from Anonymous class");
      }
    };

    hello.greet();
  }
}
```

Collection

A collection is simply an object that represents a group of objects, known as its elements.

Collection Framework

It provides a set of classes and interfaces to store, manage and manipulate groups of objects efficiently.



Interfaces in collection

Collection: The root interface for all the other collection types.

List: An ordered collection that can contain duplicate elements (e.g., ArrayList, LinkedList).

Set: A collection that cannot contain duplicate elements (e.g., HashSet, TreeSet).

Queue: A collection designed for holding elements prior to processing (e.g., Priority Queue, LinkedList when used as a queue).

Deque: A double-ended queue that allows insertion and removal from both ends (e.g., ArrayDeque).

Map: An interface that represents a collection of key-value pairs (e.g., HashMap, TreeMap).

Collection Interface

The Collection interface is the root interface of the Java Collection Framework. It is the most basic interface that defines a group of objects known as elements. The Collection interface is a part of the java.util package, and It is a parent interface that is extended by other collection interfaces like List, Set, and Queue.

Since Collection is an interface, it cannot be instantiated directly; rather, it provides a blueprint for the basic operations that are common to all collections.

The Collection interface defines a set of core methods that are implemented by all classes that implement the interface. These methods allow for basic operations such as adding, removing, and checking the existence of elements in a collection.

List Interface

The List interface in Java is a part of the java.util package and is a sub-interface of the Collection interface. It provides a way to store an ordered collection of elements (known as a sequence).

Lists allow for precise control over where elements are inserted and can contain duplicate elements.

The List interface is implemented by several classes in the Java Collection Framework, such as ArrayList, LinkedList, Vector, and Stack.

Key Features of the List Interface

Order Preservation

Index-Based Access

Allows Duplicates

ArrayList

An ArrayList is a resizable array implementation of the List interface. Unlike arrays in Java, which have a fixed size, an ArrayList can change its size dynamically as elements are added or removed. This flexibility makes it a popular choice when the number of elements in a list isn't known in advance.

Internal working

Unlike a regular array, which has a fixed size, an ArrayList can grow and shrink as elements are added or removed. This dynamic resizing is achieved by creating a new array when the current array is full and copying the elements to the new array.

When you create an ArrayList, it has an initial capacity (default is 10). The capacity refers to the size of the internal array that can hold elements before needing to resize.

Adding Elements

When we add an element to an ArrayList, the following steps occur

- **Check Capacity:** Before adding the new element, ArrayList checks if there is enough space in the internal array (elementData). If the array is full, it needs to be resized.
- **Resize if Necessary:** If the internal array is full, the ArrayList will create a new array with a larger capacity (usually 1.5 times the current capacity) and copy the elements from the old array to the new array.
- **Add the Element:** The new element is then added to the internal array at the appropriate index, and the size is incremented.

Resizing the Array

- **Initial Capacity:** By default, the initial capacity is 10. This means the internal array can hold 10 elements before it needs to grow.
- **Growth Factor:** When the internal array is full, a new array is created with a size 1.5 times the old array. This growth factor balances memory efficiency and resizing cost.
- **Copying Elements:** When resizing occurs, all elements from the old array are copied to the new array, which is an O(n) operation, where n is the number of elements in the ArrayList.

Removing Elements

- **Check Bounds:** The ArrayList first checks if the index is within the valid range.
- **Remove the Element**: The element is removed, and all elements to the right of the removed element are shifted one position to the left to fill the gap.
- **Reduce Size:** The size is decremented by 1.

Creating an ArrayList

```
// Default constructor, creates an empty ArrayList with an initial capacity of 10
ArrayList<String> list = new ArrayList<>();

// Creating an ArrayList with a specified initial capacity
ArrayList<String> listWithCapacity = new ArrayList<>(20);

// Creating an ArrayList from another collection
List<String> anotherList = Arrays.asList("Apple", "Banana", "Orange");
ArrayList<String> listFromCollection = new ArrayList<>(anotherList);
```

Modifying Elements

- list.set(1, "Grapes"); // Replaces "Orange" with "Grapes

Removing Elements

```
// Removing by index
list.remove(1); // Removes the element at index 1 ("Grapes")

// Removing by value
list.remove("Apple"); // Removes "Apple" from the list
```

Converting to Array

- String[] array = list.toArray(new String[0]);

Time Complexity

- Access by index (get) is O(1).
- Adding an element is O(n) in the worst case when resizing occurs.
- Removing elements can be O(n) because it may involve shifting elements.
- Iteration is O(n).

LinkedList

The LinkedList class in Java is a part of the Collection framework and implements the List interface. Unlike an ArrayList, which uses a dynamic array to store the elements, a LinkedList stores its elements as nodes in a doubly linked list. This provides different performance characteristics and usage scenarios compared to ArrayList.

Performance Considerations

LinkedList has different performance characteristics compared to ArrayList:

Insertions and Deletions: LinkedList is better for frequent insertions and deletions

in the middle of the list because it does not require shifting elements, as in ArrayList. +

Random Access: LinkedList has slower random access (get(int index)) compared to ArrayList because it has to traverse the list from the beginning to reach the desired index.

Memory Overhead: LinkedList requires more memory than ArrayList because each node in a linked list requires extra memory to store references to the next and previous nodes.

Vector

A Vector in Java is a part of the java.util package and is one of the legacy classes in Java that implements the List interface.

It was introduced in JDK 1.0 before collection framework and is synchronized, making it thread-safe.

Now it is a part of collection framework.

However, due to its synchronization overhead, it's generally recommended to use other modern alternatives like ArrayList in single-threaded scenarios. Despite this, Vector is still useful in certain situations, particularly in multi-threaded environments where thread safety is a concern.

Key Features of Vector

Dynamic Array: Like ArrayList, Vector is a dynamic array that grows automatically when more elements are added than its current capacity.

Synchronized: All the methods in Vector are synchronized, which makes it thread-safe. This means multiple threads can work on a Vector without the risk of corrupting the data. However, this can introduce performance overhead in single-threaded environments.

Legacy Class: Vector was part of Java's original release and is considered a legacy class. It's generally recommended to use ArrayList in single-threaded environments due to performance considerations.

Resizing Mechanism: When the current capacity of the vector is exceeded, it doubles its size by default (or increases by a specific capacity increment if provided).

Random Access: Similar to arrays and ArrayList, Vector allows random access to elements, making it efficient for accessing elements using an index.

Constructors of Vector

Vector(): Creates a vector with an initial capacity of 10.

Vector(int initialCapacity): Creates a vector with a specified initial capacity.

Vector(int initialCapacity, int capacityIncrement): Creates a vector with an initial capacity and capacity increment (how much the vector should grow when its capacity is exceeded).

Vector(Collection<? extends E> c): Creates a vector containing the elements of the specified collection.

Synchronization and Performance

Since Vector methods are synchronized, it ensures that only one thread can access the vector at a time. This makes it thread-safe but can introduce performance overhead in single-threaded environments because synchronization adds locking and unlocking costs.

In modern Java applications, ArrayList is generally preferred over Vector when synchronization isn't required. For thread-safe collections, the CopyOnWriteArrayList or ConcurrentHashMap from the java.util.concurrent package is often recommended instead.

Stack

Stack extends Vector, it is synchronized, making it thread-safe.

LIFO Structure: Stack follows the Last-In-First-Out (LIFO) principle, where the last element added is the first one to be removed.

Inheritance: Stack is a subclass of Vector, which means it inherits all the features of a dynamic array but is constrained by the stack's LIFO nature.

CopyOnWriteArrayList

CopyOnWriteArrayList class is introduced in JDK 1.5, which implements the List inteface. It is an enhanced version of **ArrayList** in which all modifications (add, set, remove, etc) are implemented by making a fresh copy. It is found in **java.util.concurrent** package. It is a data structure created to be used in a concurrent environment.

As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at a certain point both will be synchronized automatically, which is taken care of by JVM. Therefore, there is no effect for threads that are performing read operation.

It is costly to use because for every update operation a cloned copy will be created. Hence, CopyOnWriteArrayList is the best choice if our frequent operation is read operation.

It is a thread-safe version of ArrayList.

Insertion is preserved, duplicates, null, and heterogeneous Objects are allowed.

Map Interface

In Java, a Map is an object that maps keys to values. It cannot contain duplicate keys, and each key can map to at most one value. Think of it as a dictionary where you look up a word (key) to find its definition (value).

Key Characteristics of the Map Interface

Key-Value Pairs: Each entry in a Map consists of a key and a value.

Unique Keys: No two entries can have the same key.

One Value per Key: Each key maps to a single value.

Order: Some implementations maintain insertion order (LinkedHashMap), natural order (TreeMap), or no order (HashMap).

HaspMap

- **Unordered:** Does not maintain any order of its elements.
- **Allows null Keys and Values:** Can have one null key and multiple null values.
- **Not Synchronized:** Not thread-safe; requires external synchronization if used in a multi-threaded context.
- **Performance:** Offers constant-time performance (O(1)) for basic operations like get and put, assuming the hash function disperses elements properly.

Internal Working of HashMap

A hash function is an algorithm that takes an input (or "key") and returns a fixed-size string of bytes, typically a numerical value. The output is known as a hash code, hash value, or simply hash. The primary purpose of a hash function is to map data of arbitrary size to data of fixed size

Deterministic: The same input will always produce the same output.

Fixed Output Size: Regardless of the input size, the hash code has a consistent size (e.g., 32-bit, 64-bit).

Efficient Computation: The hash function should compute the hash quickly.

```
class Node<K, V> {
    final int hash;   // hash code of the key
    final K key;      // the key itself
    V value;          // the value associated with the key
    Node<K, V> next;  // pointer to the next node in case of a collision (linked list)
}
```

How Data is Stored in HashMap

Step 1: Hashing the Key

First, the key is passed through a hash function to generate a unique hash code (an integer number). This hash code helps determine where the key-value pair will be stored in the array (called a "bucket array").

Step 2: Calculating the Index

The hash code is then used to calculate an index in the array (bucket location) using

int index = hashCode % arraySize;

The index decides which bucket will hold this key-value pair.

For example, if the array size is 16, the key's hash code will be divided by 16, and the remainder will be the index.

Step 3: Storing in the Bucket

The key-value pair is stored in the bucket at the calculated index. Each bucket can hold multiple key-value pairs

(this is called a collision handling mechanism, discussed later).

How HashMap Retrieves Data

When we call get(key), the HashMap follows these steps:

- **Hashing the Key:** Similar to insertion, the key is hashed using the same hash function to calculate its hash code.
- **Finding the Index:** The hash code is used to find the index of the bucket where the key-value pair is stored.
- **Searching in the Bucket:** Once the correct bucket is found, it checks for the key in that bucket. If it finds the key, it returns the associated value.

Handling Collisions

Since different keys can generate the same index (called a collision), HashMap uses a technique to handle this situation. Java's HashMap uses Linked Lists (or balanced trees after Java 8) for this.

If multiple key-value pairs map to the same bucket, they are stored in a linked list inside the bucket.

When a key-value pair is retrieved, the HashMap traverses the linked list, checking each key until it finds a match.

map.put("apple", 50);

map.put("banana", 30);

map.put("orange", 80);

Let's say "apple" and "orange" end up in the same bucket due to a hash collision. They will be stored in a linked list in that bucket:

Bucket 5: ("apple", 50) -> ("orange", 80)

When we do map.get("orange"), HashMap will go to Bucket 5 and then traverse the linked list to find the entry with the key "orange".

HashMap Resizing (Rehashing)

HashMap has an internal array size, which by default is 16.

When the number of elements (key-value pairs) grows and exceeds a certain load factor (default is 0.75), HashMap automatically resizes the array to hold more data. This process is called rehashing.

The default size of the array is 16, so when more than 12 elements (16 * 0.75) are inserted, the HashMap will resize.

During rehashing

The array size is doubled.

1. All existing entries are rehashed (i.e., their positions are recalculated) and placed into the new array.

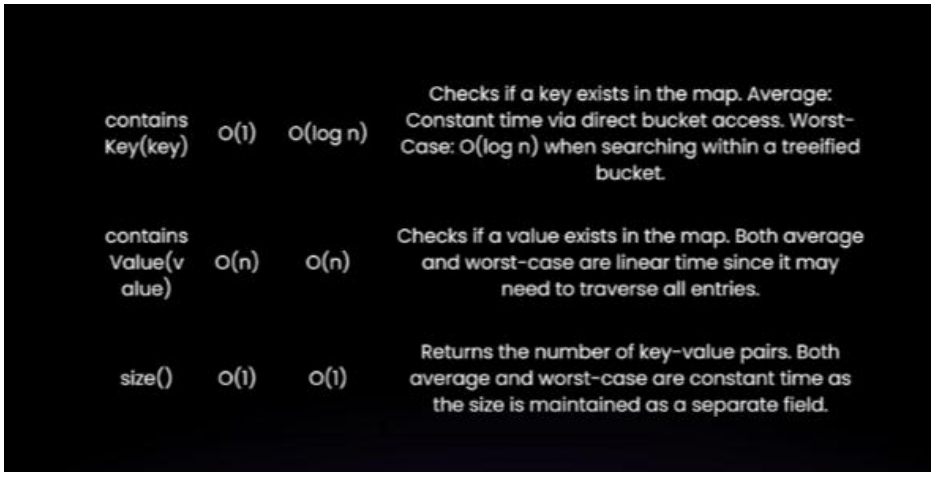
2. This ensures the HashMap continues to perform efficiently even as more data is added.

Time Complexity

HashMap provides constant time O(1) performance for basic operations like put() and get() (assuming no collisions). However, if there are many collisions, and many entries are stored in the same bucket, the performance can degrade to O(n), where n is the number of elements in that bucket.

But after Java 8, if there are too many elements in a bucket, HashMap switches to a balanced tree instead of a linked list to ensure better performance O(log n).

Operation	Average-Case Time Complexity	Worst-Case Time Complexity	Explanation
put(key, value)	O(1)	O(log n)	Inserts a key-value pair. Average: Constant time due to direct bucket access. Worst-Case: O(log n) when bucket converts to a Red-Black Tree after exceeding collision threshold.
get(key)	O(1)	O(log n)	Retrieves the value associated with a key. Average: Constant time via direct bucket access. Worst-Case: O(log n) when searching within a treeified bucket.
remove(key)	O(1)	O(log n)	Removes the key-value pair associated with a key. Average: Constant time with direct access. Worst-Case: O(log n) when removing from a treeified bucket.



LinkedHashMap

LinkedHashMap in Java is a subclass of HashMap that maintains the **insertion order** (or optionally **access order**) of its entries. Here's everything you need to know for interviews:

✅ What is LinkedHashMap?

- It is an implementation of the Map interface that combines:
  - **Hashing** (like HashMap) for fast lookups.
  - **Linked list** to maintain predictable iteration order.
- Preserves **order of keys**:
  - **Insertion order** by default.
  - **Access order** if configured (useful for LRU caches).

Order Maintenance

- When you insert a new entry, it is added at the **end of the linked list**.
- If accessOrder = true, every time you call get() or put() on an existing key, that entry moves to the end (used for LRU).

WeakHashMap

WeakHashMap in Java is a special implementation of the Map interface that uses **weak references** for its keys. Here's a complete breakdown:

✅ What is WeakHashMap?

- A WeakHashMap<K, V> is similar to a normal HashMap, but the **keys are stored as weak references**.
- If a key is **no longer referenced elsewhere in the application**, the garbage collector (GC) can reclaim it, and the corresponding entry is automatically removed from the map.

🔍 Internal Working

1. **Weak Reference for Keys**
    - Internally, keys are wrapped in WeakReference objects.
    - This means the GC does not consider these keys as strongly reachable.
2. **Garbage Collection Impact**
    - When GC runs and finds that a key has no strong references, it clears the weak reference.
    - The entry is then removed from the map during the next access or cleanup.
3. **Values**
    - Values are stored as **normal strong references**.
    - Only keys are weakly referenced.

✅ Use Case

- Useful for **caching** where you don't want cached objects to prevent GC from reclaiming memory.
- Example: Image cache, where if the image is not used anywhere else, it should be removed automatically.

ncxb