TELUSKO

10.1 Abstract Keyword

Introduction

In Java, the abstract keyword is a non-access modifier used with classes and methods (but not variables) to achieve **abstraction**, one of the four pillars of **Object-Oriented Programming (OOP)**. Abstraction allows us to define a template or blueprint without requiring a full implementation. It's mainly used to hide the complex implementation details and show only the essential features of an object.

Abstract Methods in Java

Sometimes, you may only want to declare a method in a superclass without providing its implementation. This can be achieved using the abstract keyword. Such methods are often referred to as **subclass responsibilities**, as the subclass must provide their implementation. The superclass only specifies that the method exists but does not define its behaviour.

To declare an abstract method, use the following syntax:

```
abstract return_type methodName(parameters);
```

Key Characteristics:

- Abstract methods have no body; they only include the method signature, followed by a semicolon.
- A subclass that extends an abstract class **must** override and provide the implementation for all abstract methods; otherwise, the subclass will also be abstract.

Abstract Classes in Java

An **abstract class** is a class that may contain abstract methods (methods without implementation) as well as concrete methods (methods with implementation). You cannot instantiate an abstract class directly; it serves as a blueprint for other classes.

To declare an abstract class, you use the abstract keyword:

TELUSKO

```
abstract class ClassName {
    // Abstract methods
    abstract void methodName();

    // Concrete methods
    void concreteMethod() {
        // method body
    }
}
```

Example:

```
abstract class Car {
    // Abstract method
    public abstract void drive();

    // Concrete method
    public void playMusic() {
        System.out.println("Playing music...");
    }
}
```

Characteristics of the Abstract Keyword

The abstract keyword in Java is mainly used to define abstract classes and methods. Here are some key characteristics:

- **Cannot Instantiate Abstract Classes:** You cannot create an object of an abstract class. The abstract class is meant to be extended by concrete (non-abstract) classes, which implement the abstract methods.
- **Abstract Methods Lack Implementation:** Abstract methods only provide a method signature and no implementation. The subclasses are responsible for defining these methods.
- **Abstract Classes Can Have Both Abstract and Concrete Methods:** Abstract classes may include concrete methods (methods with bodies) alongside abstract ones.
- **Abstract Classes Can Have Constructors:** Though you cannot instantiate abstract classes, they can still have constructors. These constructors are used by subclasses to initialize the inherited fields.

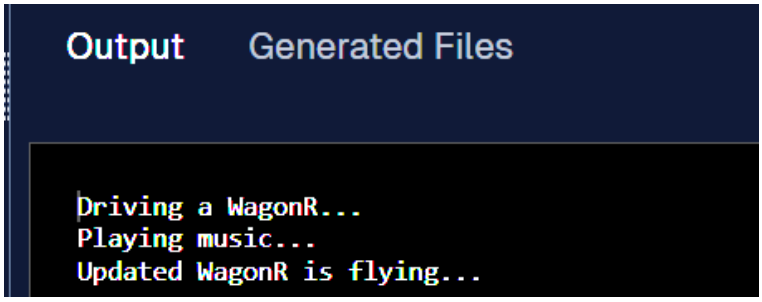
- **Can Contain Instance Variables:** Abstract classes can include instance variables, which can be accessed by both the abstract class and its subclasses.
- **Can Implement Interfaces:** Abstract classes can implement interfaces and must provide implementations for all interface methods unless the class itself is abstract.

Real-World Example

Let's consider a scenario where we want to create a blueprint for different types of cars:

```
abstract class Car {
    // Abstract method (no implementation)
    public abstract void drive();

    // Abstract method (no implementation)
    public abstract void fly();

    // Concrete method
    public void playMusic() {
        System.out.println("Playing music...");
    }}
// Concrete subclass of Car
abstract class WagonR extends Car {
    // Implementing abstract method drive
    @Override
    public void drive() {
        System.out.println("Driving a WagonR...");
    }

    // Keep fly as abstract
    @Override
    public abstract void fly();
}
// Concrete subclass of WagonR
class UpdatedWagonR extends WagonR {
    // Implementing the fly method
    @Override
    public void fly() {
        System.out.println("Updated WagonR is flying...");
    }}
public class MyClass {
  public static void main(String args[]) {
      // No need for typecasting, using dynamic dispatch
        Car car = new UpdatedWagonR();
        car.drive();      // Calls the drive() method from WagonR
        car.playMusic();  // Calls the concrete method from Car class
        car.fly();        // Calls the fly() method from UpdatedWagonR
  }}
```

Output:



Summary:

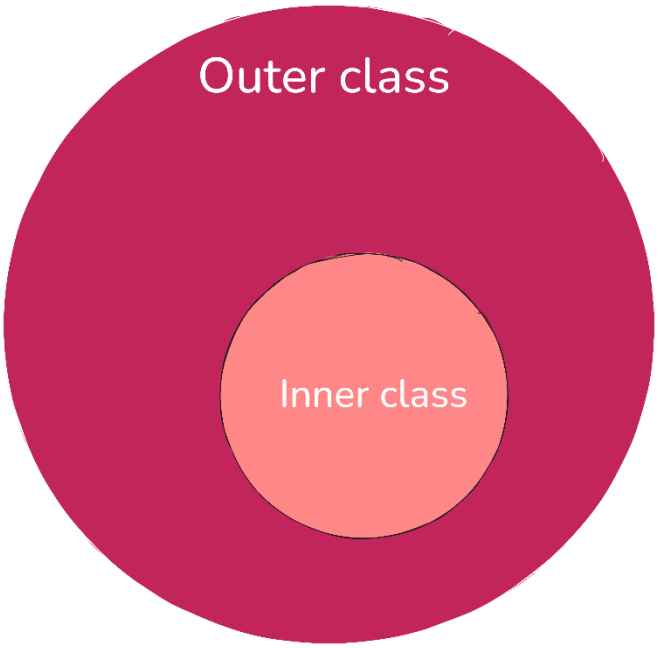
Here are the key takeaways regarding the abstract keyword in Java:

1. **Abstract Classes Cannot Be Instantiated:** You cannot create an object of an abstract class directly.
2. **Abstract Methods:** These methods are declared but not defined; they must be implemented in subclasses.
3. **Abstract Class Requirement:** If a class contains at least one abstract method, the class itself must be declared abstract.
4. **Concrete Methods in Abstract Classes:** An abstract class can contain both abstract and concrete methods.
5. **Abstract Class Constructors:** Abstract classes can have constructors, but they are only used by subclasses.

• 10.2-Inner Class

➢ Introduction

In Java, an **inner class** refers to a class that is declared **inside another class**. Inner classes were primarily introduced to group logically related classes together, reflecting the object-oriented nature of Java and its attempt to mirror real-world relationships.



➢ Syntax of Inner Class

The general syntax for defining an inner class is as follows:

```
class OuterClass {
    // Outer class code

    class InnerClass {
        // Inner class code
    }
}
```

➢ Advantages of Inner Classes

Using inner classes has several advantages:

- **Encapsulation**: It allows better encapsulation of the code, especially when the inner class is logically dependent on the outer class.
- **Access to Outer Class Members**: Inner classes can access the private members (fields and methods) of the outer class, providing an added dimension to the class design.
- **Cleaner Code**: It leads to more modular, readable, and optimized code.

➢ Example: Basic Inner Class

```
class A {
    public void show() {
        System.out.println("in Show");
    }

    class B {
        public void config() {
            System.out.println("in Config");
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new A(); // Creating an instance of the outer class
        obj.show();      // Calling method of the outer class

        // Creating an instance of the inner class using the outer class object
        A.B obj1 = obj.new B();
        obj1.config();   // Calling method of the inner class
    }
}
```

Output:

```
Output    Generated Files

in Show
in Config
```

Explanation

In this example:

- We have two classes, A (outer class) and B (inner class).
- Class A has a method show(), and class B has a method config().
- To create an object of the inner class B, we first need to create an object of the outer class A. This is because the inner class is dependent on the outer class.

➢ **Code Explanation**:

```
A.B obj1 = obj.new B();
```

Here, A.B indicates that B is an inner class of A, and obj.new B() uses the object obj of A to create an instance of B.

After executing we get two class files one of class A and the other Other of class B.

Here, B class file is created with the name **A$B** where $ represents B class is inner class and it belongs to class A.

➢ **Inner Class Dependency**

An inner class is usually tightly coupled with the outer class. Without the outer class, the inner class often makes no sense. In the example above, **class B depends on class A** for its existence.

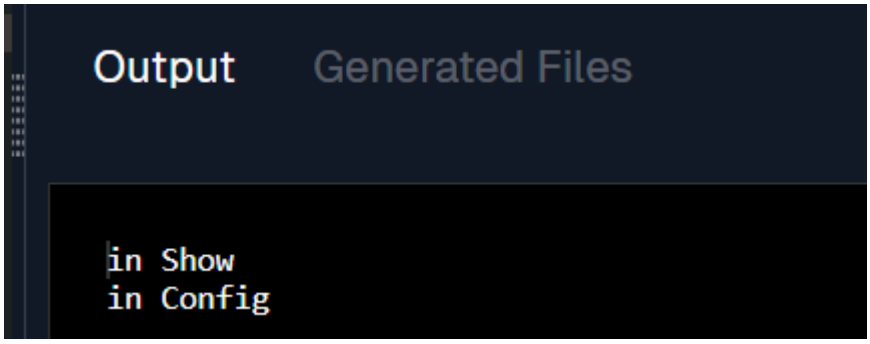
➢ **Static Inner Class**

Inner classes can also be marked as static. A **static inner class** is independent of the outer class and can be accessed without creating an object of the outer class.

➢ Example: Static Inner Class

```java
class A {
    public void show() {
        System.out.println("in Show");
    }

    static class B {
        public void config() {
            System.out.println("in Config");
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new A(); // Creating an instance of the outer class
        obj.show();      // Calling method of the outer class

        // Creating an instance of the static inner class without an outer class object
        A.B obj1 = new A.B();
        obj1.config();   // Calling method of the static inner class
    }
}
```

Output:



➢ Key Points for Static Inner Class:

- **Independent of the Outer Class**: A static inner class can be instantiated without an object of the outer class.
- **Static Context**: The static inner class behaves like a regular class and cannot access the non-static members of the outer class directly.
- **Rules for Inner and Outer Classes**

- **Static Modifier**: Only inner classes can be marked as static. Outer classes cannot be declared as static, and any attempt to do so will result in a compile-time error.
- **Allowed Modifiers for Outer Class**: The valid access modifiers for an outer class are public, final, and abstract.

➢ Summary

- Inner classes are declared within another class.
- An inner class can access private members of its outer class.
- To instantiate an inner class, an object of the outer class is required.
- A static inner class can be instantiated without creating an object of the outer class.
- Static modifier applies only to inner classes, not outer classes.

10.3-Anonymous Inner class

Definition:
In Java, an anonymous inner class is a type of inner class without a name. It is used to create an object with certain modifications or behaviours without explicitly declaring a subclass. This feature is commonly used when you need a one-time use implementation for a class or an interface.

Key Points:

- **Anonymous** means without a name.
- **Anonymous inner classes** are created without naming the class.
- They are typically used to extend a class or implement an interface.

Syntax

The syntax for creating an anonymous inner class involves the following:

```
new ClassName() {
    // method overriding or additional methods
};
```

Example 1: Without Anonymous Inner Class

```
class A {
    public void show() {
        System.out.println("In A's show method");
    }
}

class B extends A {
    @Override
    public void show() {
        System.out.println("In B's show method");
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new B();  // B class object
        obj.show();
    }
}
```

Output:

```
In B's show method
```

Explanation:

- In this example, we have a class A with a method show().
- Class B extends A and overrides the show() method.
- The object obj is created for class B, which results in the overridden method show() being called.
- This works but requires creating a separate class (B) even if it's only used once.

Example 2: Using Anonymous Inner Class

```
class A {
    public void show() {
        System.out.println("In A's show method");
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new A() {
            @Override
            public void show() {
                System.out.println("In new anonymous show method");
            }
        };
        obj.show();
    }
}
```

Output:

```
In new anonymous show method
```

Explanation:

- Instead of creating a separate subclass like B, we use an anonymous inner class.

- The curly braces {} after new A() define the anonymous inner class, where we override the show() method.
- This creates a class without a name, allowing a one-time customization of A.

Advantages of Anonymous Inner Classes:

1. **Encapsulation**: Keeps implementation details close to where they are used, improving code organization.
2. **Access Control**: Can access private members of the outer class, even if they are private.
3. **Code Optimization**: Avoids the need to create a separate class, reducing boilerplate code.
4. **Polymorphism**: Provides a dynamic way to override methods at runtime.
5. **Reduced Code Complexity**: Simplifies the code when you need minor changes to a class or interface just once.

When to Use:

- When you need to override methods of a class or interface just once.
- When a specific implementation is required without creating a new, named class.

Key Features of Anonymous Inner Class:

- **No name**: It is defined without any class name.
- **Single Use**: It's typically used for one-time code modifications.
- **Extends or Implements**: You can extend a class or implement an interface on the fly.

TELUSKO

10.4 - Abstract Class and Anonymous Inner Class

Introduction:

In Java, an **anonymous inner class** allows you to create an instance of a class with new behaviour without having to explicitly define a subclass. This is particularly useful when you need to modify or extend the functionality of an existing class only once, saving the effort of defining a new class.

On the other hand, an **abstract class** is a class marked with the abstract keyword, which means:

- It may contain both **abstract methods** (without implementations) and **concrete methods** (with implementations).
- You **cannot create an object** of an abstract class directly.

But can we combine the use of abstract classes with anonymous inner classes? The answer is yes! Let's explore this combination with an example.

Example: Combining Abstract Class and Anonymous Inner Class

```
abstract class A {
    // Abstract method
    public abstract void show();
}

public class Demo {
    public static void main(String[] args) {
        A obj = new A() {
            // Implementation for abstract method in anonymous inner class
            @Override
            public void show() {
                System.out.println("In new show");
            }
        };
        obj.show(); // Calling the overridden method
    }
}
```

TELUSKO

Output:

```
In new show
```

Explanation:

In this example:

- A is an **abstract class** that contains an abstract method show().
- Even though you cannot create an object of A directly because it is abstract, we are able to provide an implementation of the show() method using an **anonymous inner class**.
- The anonymous inner class gives us the ability to **override the abstract method** and provide a new implementation.

This demonstrates that:

- **Anonymous inner classes** can implement abstract methods, even for abstract classes.
- You can combine both concepts to dynamically implement abstract methods without having to define a separate subclass.

Additionally, this approach works for **multiple abstract methods** in an abstract class, allowing you to implement several methods within an anonymous inner class.

Key Points to Remember:

- **Abstract classes** can't be instantiated directly, but **anonymous inner classes** allow you to create an instance that implements abstract methods.
- This combination is particularly useful when you need a **one-time implementation** of an abstract class without creating a named subclass.
- The implementation of abstract methods inside an anonymous inner class is done inline and is **not reusable** elsewhere, making it ideal for short, specific tasks.

10.5-What is Interface

What is an Interface in Java?

In Java, the abstract keyword is used at the class level to make a class abstract. An abstract class can contain methods without implementation, called **abstract methods**, as well as methods with implementation, called **concrete methods**. However, when a class has only abstract methods, an alternative to using an abstract class to achieve abstraction we use an **interface**.

By using interface we can achieve 100% abstraction.

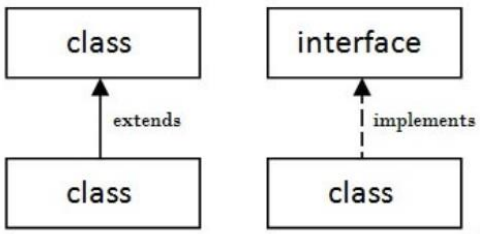
What is an Interface?

An **interface** in Java is a blueprint of a class that can have static constants and abstract methods. It is a mechanism to achieve **abstraction** and **multiple inheritance** in Java.

- Interfaces **cannot have method bodies**; they only declare methods.
- You cannot instantiate an interface directly, similar to abstract classes.
- Methods declared in an interface are, by default, marked as **public** and **abstract**.
- Variables in an interface are implicitly **public**, **static**, and **final**.

Key Features of an Interface

- **Total Abstraction:** All methods in an interface are abstract (i.e., they have no implementation). This means that the class implementing the interface must provide implementations for all declared methods.
- **IS-A Relationship:** Interfaces represent the "IS-A" relationship in Java, similar to inheritance.
- **Implements Keyword:** A class uses the **implements** keyword to indicate that it provides implementations for the interface methods.



- **Static and Final Variables:** Variables declared in an interface are static and final by default, meaning they are constants that cannot be changed once initialized.

Syntax of an Interface

To declare an interface, the interface keyword is used. Here's the syntax for declaring an interface:

```
interface InterfaceName {
    // Declare constant fields
    // Declare abstract methods
}
```

Example:

```
interface A {
    void show();  // implicitly public and abstract
    void config();
}
```

Instantiating an Interface

You cannot directly create an object of an interface, just like an abstract class. The following code will result in an error:

```
A obj = new A();  // Error: Cannot instantiate the type A
```

However, you can instantiate a class that implements the interface, as shown in the example below.

Example of Interface Implementation

```java
interface A {
    int age = 44;  // final and static by default
    String area = "Mumbai";

    void show();
    void config();
}

class B implements A {
    public void show() {
        System.out.println("In show");
    }

    public void config() {
        System.out.println("In config");
    }
}

public class MyClass {
  public static void main(String args[]) {
    A obj = new B();  // Instantiate a class that implements the interface
        obj.show();
        obj.config();

        // Accessing final and static variables of the interface
        System.out.println(A.age);
        System.out.println(A.area);
  }
}
```

Explanation of Interface Usage

An interface can be considered a **contract** between the class and the interface itself. It defines **unimplemented methods** that the class must provide implementations for. In essence, an interface lays down the **requirements** (methods and constants), which the implementing class must fulfill.

In software design, interfaces are sometimes referred to as **Service Requirement Specifications (SRS)** because they specify the services or methods that classes need to implement.

Important Points About Interfaces in Java

- Interfaces are not classes but provide a structure similar to classes.
- Methods in interfaces are always public and abstract.
- Variables declared in interfaces are constants by default (public, static, final).
- A class that implements an interface must provide implementations for all its methods.
- A class can implement multiple interfaces, thus achieving multiple inheritance.

10.7-More on Interfaces

In previous sections, we've explored interfaces in Java in depth. Now, let's dive into more advanced topics regarding interfaces. A common question is: **Can a class implement multiple interfaces?**

Multiple Interface Implementation: Example

In Java, a class can implement multiple interfaces. This is unlike class inheritance, where a class can only extend one class. Implementing multiple interfaces allows for a more flexible design.

Here's an example:

```
interface A {
    int age = 44;
    String area = "Mumbai";
    void show();
    void config();
}
interface X {
    void run();
}
interface Y {
    // Empty interface
}
class B implements A, X {

    public void show() {
        System.out.println("in show");
    }

    public void config() {
        System.out.println("in config");
    }

    public void run() {
        System.out.println("in run");
    }
}
public class MyClass {
  public static void main(String args[]) {
        A obj;
        obj = new B();  // Using A's reference
        obj.show();
        obj.config();
        // The variables in interfaces are final and static,
        // so we can access them directly using the interface name
        System.out.println(A.age);
        System.out.println(A.area)
        // obj.run(); // Error: Cannot call run() through A's refe
        X obj1;
        obj1 = new B(); // Using X's reference
        obj1.run(); }}
```

Output:

```
in show
in config
44
Mumbai
in run
```

Explanation

- **Interface A**: Contains two abstract methods show() and config(), which are implemented by class B.
- **Interface X**: Contains one abstract method run(), also implemented by class B.
- **Interface Y**: This is an empty interface (also called a marker interface).

In Java, a class can only extend **one class**, but it can implement **multiple interfaces**. In this example, class B implements both A and X.

- The variables age and area in interface A are both **final** and **static**, which means they can be accessed directly through the interface name, without creating an object.

Multiple Inheritance Through Interfaces

```
interface A {
    int age = 44;
    String area = "Mumbai";
    void show();
    void config();
}
interface X {
    void run();
}
interface Y extends X {
        // Inherits the run() method from X
}
class B implements A, Y {

    public void show() {
        System.out.println("in show");
    }

    public void config() {
        System.out.println("in config");
    }

    public void run() {
        System.out.println("in run");
    }
}
public class MyClass {
  public static void main(String args[]) {
        A obj;
        obj = new B();  // Using A's reference
        obj.show();
        obj.config();
        // obj.run(); // Error: Cannot call run() through A's refere
        Y obj1;
        obj1 = new B(); // Using X's reference
        obj1.run(); }}
```

Output:

```
in show
in config
in run
```

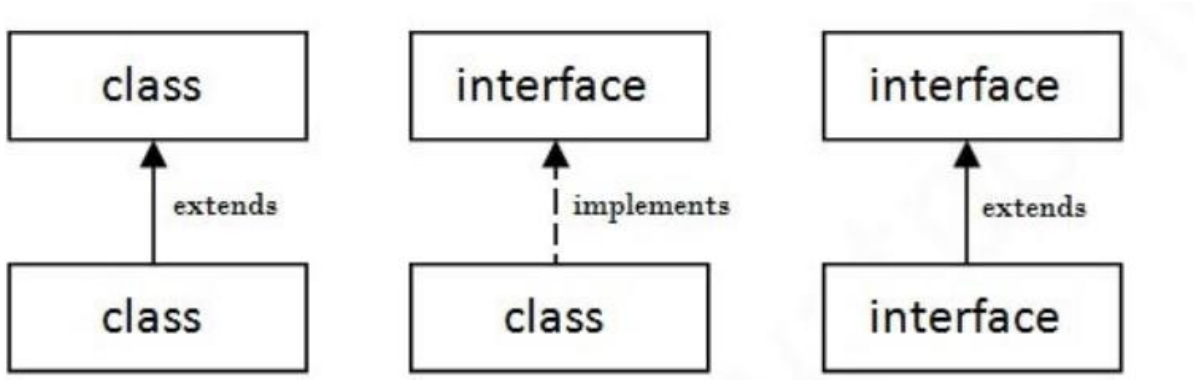
Let's extend the previous example with multiple interface inheritance. Suppose we have interface X, and now Y extends X. The run() method will automatically be inherited by Y, and class B will need to implement it.

Explanation

- **Interface Y extends X**: Y inherits the run() method from X. When class B implements Y, it must still implement the run() method because Y inherits it from X.
- In the Demo class, if we use a reference of type Y, we can call the run() method without issues.

Summary of Key Concepts

- **Class to Class**: Use the extends keyword.
- **Class to Interface**: Use the implements keyword.
- **Interface to Interface**: Use the extends keyword.



A class can implement multiple interfaces at the same time. In the example above, class B implements both A and Y (which extends X). The run() method must be implemented in class B because Y inherits this method from X.

10.6-Need of Interface

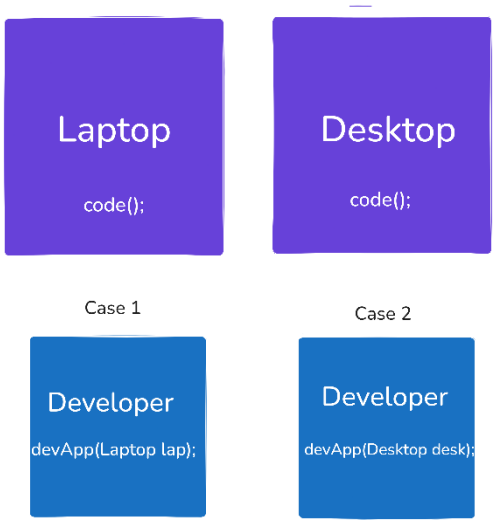
In this section, we'll explore the importance of interfaces in Java and understand their usage with examples. Interfaces play a crucial role in achieving abstraction, flexibility, and adhering to design principles like **"programming to an interface"**.

Initial Scenario: Without Interface

Consider the following example:

```
class Laptop {
    public void code() {
        System.out.println("Code, compile, and run");
    }
}

class Desktop {
    public void code() {
        System.out.println("Code, compile, and run... faster");
    }
}

class Developer {
    public void devApp(Laptop laptop) {
        laptop.code();
    }
}


public class MyClass {
  public static void main(String args[]) {
    Laptop laptop = new Laptop();
        Desktop desktop = new Desktop();
        Developer developer = new Developer();
        developer.devApp(laptop); //Output : Code, compile, and run

  }
}
```

Explanation:



In this example, there are three main classes:

1. **Laptop** – Represents a laptop that can run code.
2. **Desktop** – Represents a desktop that runs code faster than a laptop.
3. **Developer** – The developer class that uses the device (either Laptop or Desktop) to write and run code.

The Developer class has a method devApp(Laptop laptop), which takes a Laptop object as a parameter and calls its code() method. In the Demo class, we instantiate the Laptop, Desktop, and Developer classes, and pass the Laptop object to the developer.

Problem:

Initially, the developer works with a laptop. But later, if the company decides to provide a desktop instead, changes must be made to the Developer class to accommodate the desktop. This violates the design principle of flexibility. Each time the working environment changes, we would have to modify the Developer class.

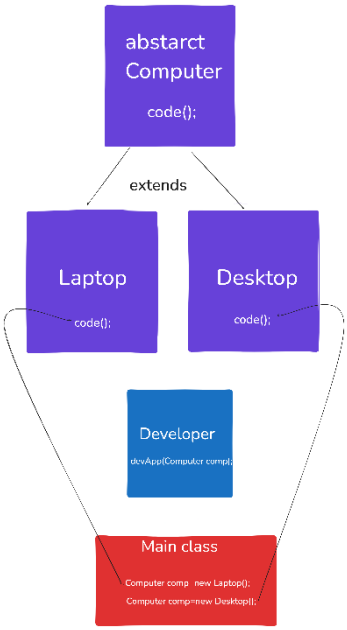
This is where **interfaces** come into play.

Solution: Using Abstraction

One possible improvement is to use **abstraction** to generalize both the Laptop and Desktop classes. Both are essentially computers, so we can introduce an abstract class called Computer:

```java
abstract class Computer {
    public abstract void code();
}

class Laptop extends Computer {
    public void code() {
        System.out.println("Code, compile, and run");
    }
}

class Desktop extends Computer {
    public void code() {
        System.out.println("Code, compile, and run... faster");
    }
}

class Developer {
    public void devApp(Computer comp) {
        comp.code();
    }
}

public class MyClass {
  public static void main(String args[]) {
            Computer laptop = new Laptop();
        Computer desktop = new Desktop();
        Developer developer = new Developer();
        developer.devApp(laptop); // Output: "Code, compile, and run"
        developer.devApp(desktop); // Output: "Code, compile, and run... faster"

  }
}
```

Explanation:



- We created an abstract class Computer with an abstract method code().
- Both Laptop and Desktop classes extend the Computer class and provide their specific implementation of the code() method.
- The Developer class now works with the Computer type instead of a specific Laptop or Desktop. This makes the developer's code flexible and allows the developer to work with any type of computer without modifying the Developer class.

In this case:

- If a laptop is passed to devApp(), it outputs: **"Code, compile, and run"**.
- If a desktop is passed, it outputs: **"Code, compile, and run... faster"**.

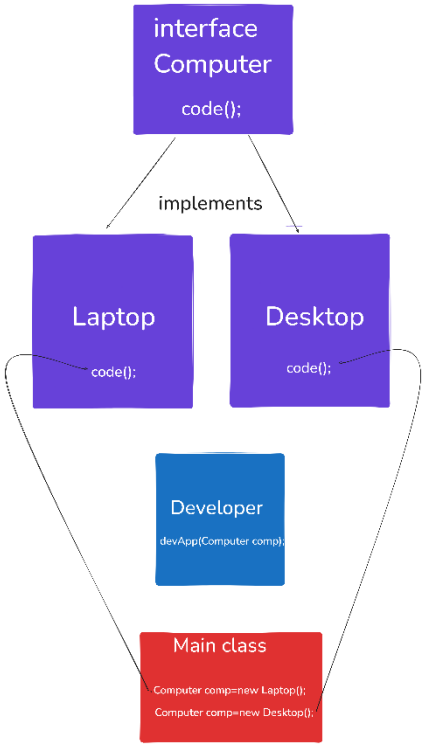
Issue with Abstract Classes:

While this solution is better, **abstract classes** only provide partial abstraction. We still need to define common code (if any) in the abstract class. If the scenario requires **full abstraction** (where only method declarations are needed without any method bodies), we should use **interfaces**.

Better Solution: Using Interface

Java interfaces provide a higher level of abstraction than abstract classes. They allow us to declare methods without providing any implementation. Let's update the code to use an interface instead of an abstract class:

```
interface Computer {
    void code();
}

class Laptop implements Computer {
    public void code() {
        System.out.println("Code, compile, and run");
    }
}

class Desktop implements Computer {
    public void code() {
        System.out.println("Code, compile, and run... faster");
    }
}

class Developer {
    public void devApp(Computer comp) {
        comp.code();
    }
}

public class MyClass {
  public static void main(String args[]) {
        Computer laptop = new Laptop();
        Computer desktop = new Desktop();
        Developer developer = new Developer();
        developer.devApp(laptop); // Output: "Code, compile, and run"
        developer.devApp(desktop); // Output: "Code, compile, and run... faster"


  }
}
```



Why Use Interfaces?

- **Full Abstraction**: All methods in an interface are abstract by default (in older versions of Java), meaning the implementing classes must provide the method body. This ensures flexibility.
- **Decoupling**: The Developer class no longer depends on the concrete implementations (Laptop or Desktop) but only on the abstraction (Computer interface). This makes the code more modular and extensible.

Conclusion:

By using an interface, we provide a flexible design where:

- We don't need to change the Developer class if a new type of computer (e.g., Tablet) is introduced.
- The code becomes more maintainable and adheres to good design principles, such as **programming to an interface**.

10.8 - What is Enum

In Java, an **enum** is essentially a special type of class that contains a fixed set of constants. Enums are often used in cases where we know all possible values at compile time. Examples include:

- Days of the week (SUNDAY, MONDAY, TUESDAY, etc.)
- Directions (NORTH, SOUTH, EAST, WEST)
- Status codes (RUNNING, FAILED, SUCCESS, etc.)

Enums are particularly useful for situations where a variable can only have one of a small set of predefined values.

Real-world Example of Enums

Consider how websites return error codes like 404 Page Not Found or 500 Internal Server Error. These codes are constants used to indicate specific conditions, much like Java's enums. In Java, enums allow us to define custom constants for scenarios like:

- Days of the week
- Status codes
- Seasons (WINTER, SPRING, SUMMER, FALL)

Defining an Enum in Java

You can define an enum using the enum keyword. For example:

```
enum Status {
    RUNNING, FAILED, PENDING, SUCCESS;
}
```

Here, the Status enum has four constants: RUNNING, FAILED, PENDING, and SUCCESS.

Example: Using Enums in Code

```
public class Demo {
    public static void main(String[] args) {
        Status status = Status.RUNNING;
        System.out.println(status); // Output: RUNNING
    }
}
```

In this example, the status variable is of type Status (an enum), and it holds the value RUNNING. You can think of Status.RUNNING as an object representing the RUNNING constant.

Characteristics of Java Enums

- **Named constants:** Enum constants are implicitly public, static, and final. This means they are constants that cannot be changed.
- **Object-oriented:** Although enums look simple, they are full-fledged objects in Java. You can define constructors, instance variables, methods, and even implement interfaces in enums.
- **Indexing:** Enum constants have an implicit order, starting from 0. You can retrieve the index of an enum constant using the ordinal() method.

Methods in Enums

Java provides several useful methods for enums:

1. **ordinal() Method**
   This method returns the index of the enum constant. The first constant is assigned an index of 0, the second 1, and so on.

   System.out.println(Status.RUNNING.ordinal()); // Output: 0

2. **values() Method**
   This method returns an array of all enum constants. It is helpful for iterating through all possible values.

Example:

```
public class Demo {
    public static void main(String[] args) {
        Status[] statuses = Status.values();
        for (Status s : statuses) {
            System.out.println(s + " at index " + s.ordinal());
        }
    }
}
```

Output:

```
RUNNING at index 0
FAILED at index 1
PENDING at index 2
SUCCESS at index 3
```

Conclusion

Enums in Java are powerful and versatile tools. They allow you to define a set of named constants, making code more readable and maintainable. With methods like **ordinal()** and **values(),** enums offer additional functionality that can be very useful when managing predefined constants.

10.9-Enum If and Switch

Java provides a special data type called **Enum**, typically used to define a collection (or set) of constants. To be more precise, an Enum is a special form of a Java class. An Enum can contain constants, attributes, methods, and more. It is commonly used when you have a fixed set of related constants, such as days of the week, directions, or statuses.

Key Features of Enum in Java:

- By default, **Enum constants are public, static, and final**.
- Enum constants are accessible using dot (.) syntax.
- Enum classes can also contain **attributes and methods**.
- Enum classes **cannot inherit other classes**, and you **cannot create objects** of Enum types.
- Enum classes can **implement interfaces**.
- Since Enums are named constants, you can compare them easily using == or control statements like **if-else** and **switch**.

Example of Enum in Java:

```
enum Status {
    Running, Failed, Pending, Success;
}
```

Here, Status is an Enum with four constants: Running, Failed, Pending, and Success.

Using Enum with if-else Statements

You can compare Enum constants using if-else statements just like primitive data types. For example:

```
public class Demo {
    public static void main(String[] args) {
        Status s = Status.Running;

        // Using if-else for comparing Enum constants
        if (s == Status.Running) {
            System.out.println("All good");
        } else if (s == Status.Failed) {
            System.out.println("Try Again");
        } else if (s == Status.Pending) {
            System.out.println("Please wait");
        } else {
            System.out.println("Done");
        }
    }
}
```

Output:

```
All good
```

In this example, the if-else statement compares the Enum constant s with the different possible statuses and prints the corresponding message.

Using Enum with Switch Statements

The switch statement is a more concise way to handle multiple conditions compared to if-else. It's especially well-suited for comparing Enum constants because the case values can directly use the Enum constants, without needing to write the Enum type.

Syntax of Switch Statement:

When you have multiple options and need to perform different actions based on those options, a switch statement is helpful. Here's how it works:

- The switch statement compares the value of a variable with several possible cases.
- Each case contains a distinct block of code.
- A break statement is typically used to exit the switch block once a match is found, although it's not mandatory.

Using Enum with Switch:

```
enum Status {
    Running, Failed, Pending, Success;
}
public class MyClass {
  public static void main(String args[]) {
    Status s = Status.Running;

        // Using switch for comparing Enum constants
        switch (s) {
            case Running:
                System.out.println("All good");
                break;

            case Failed:
                System.out.println("Try again");
                break;

            case Pending:
                System.out.println("Please wait");
                break;

            default:
                System.out.println("Done");
                break;
        }
  }
}
```

Output:

```
All good
```

In this example, the switch statement compares the Enum constant s and executes the corresponding block of code. Unlike the if-else structure, the switch statement provides a cleaner and more readable approach.

Advantages of Using Switch with Enums:

- **Readability**: The switch statement improves readability, especially when dealing with multiple conditions.
- **Clarity**: It avoids the clutter of multiple else-if conditions, making the code clearer and easier to maintain.
- **Efficiency**: In some cases, the switch statement can be faster as it's designed for comparing discrete values like Enums.

10.10-Enum Class

In Java, an **enum** is essentially a class, but it cannot be extended like a regular class. Despite this, the enum class possesses many of the properties that a normal class has. For example, we can:

- Create methods
- Define constructors
- Declare and define variables

Let's explore enums further, focusing on inheritance, constructors, and methods.

Enum and Inheritance

Although enums in Java cannot extend any other class, they implicitly extend the java.lang.Enum class. This is why enums cannot extend any other class (since Java doesn't support multiple inheritance with classes).

Key points regarding inheritance with enums:

- Enums inherit from the Enum class, which is a direct subclass of the Object class.
- The toString() method is overridden in the Enum class, returning the name of the enum constant.
- Enums **can** implement interfaces, allowing enums to behave like normal classes in that respect.

Enum and Constructors

Enums can have constructors, which are called separately for each enum constant at the time the enum is loaded. There are some specific behaviors regarding constructors in enums:

- **Private Constructors:** Enum constructors are always private. Even though you can define constructors, you cannot instantiate enums directly using the new keyword. The enum constants are the only instances of the enum type.

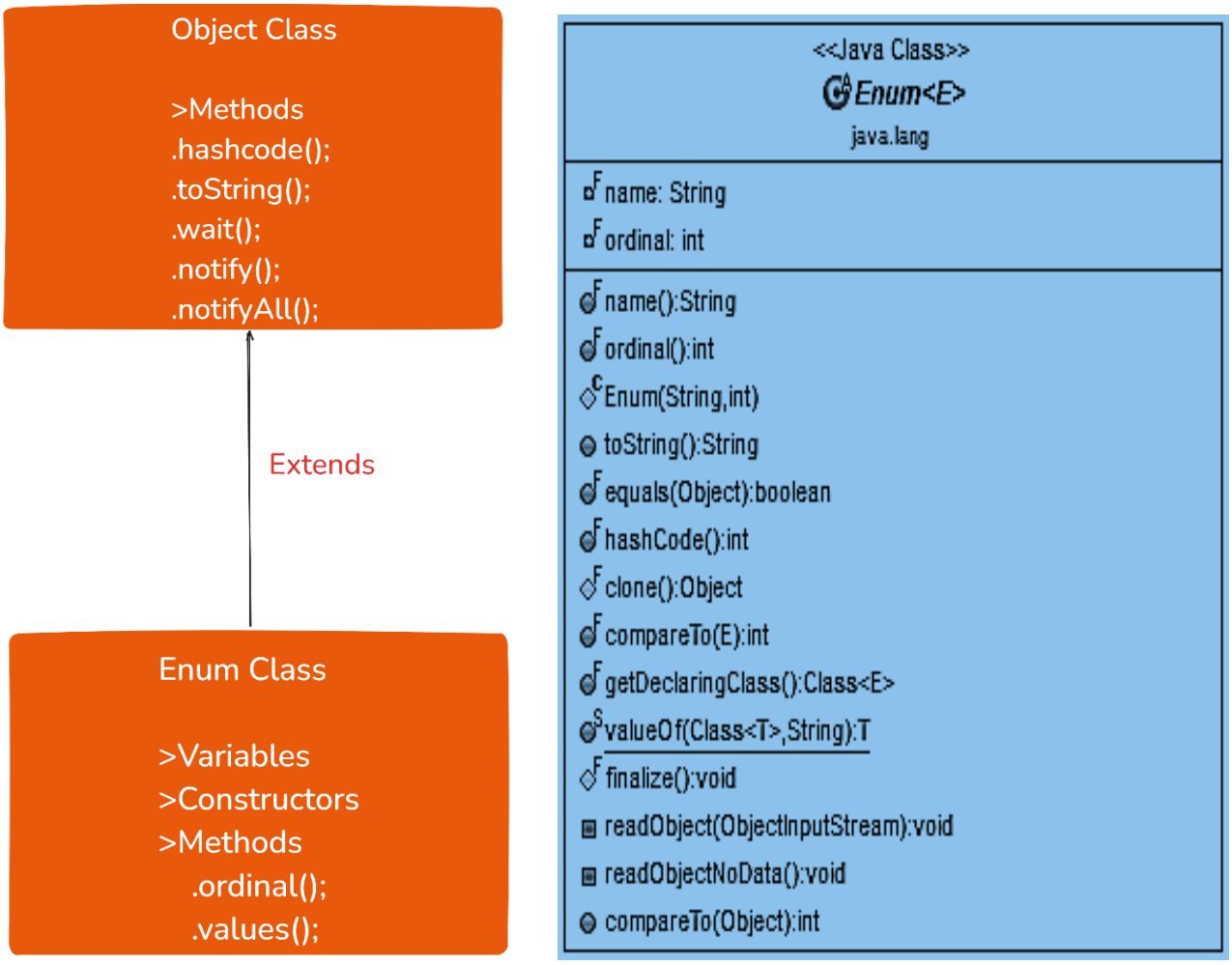
- **Parameterized Constructors:** You can use constructors to assign values to enum constants. For example, you might want to assign a price to a laptop brand in an enum.

Important: If you define a parameterized constructor, all enum constants must be initialized with corresponding values. If you omit values, you will get a compilation error.

Methods in Enum

Enums have several built-in methods, many of which come from the Enum class. Some common methods include:

- **ordinal()** – Returns the position of the enum constant.
- **values()** – Returns an array of all enum constants.



Enums can also have user-defined methods, both concrete and abstract. If you include abstract methods in an enum, each enum constant must provide an implementation for them.

Example of Enum with Constructors and Methods

Let's look at an example to illustrate how enums work, including how to use constructors and methods in enums.

```
enum Laptop {
    Macbook(2000), XPS(2200), Surface(2200),
ThinkPad(1500);
    private int price;

    // Parameterized constructor
    private Laptop(int price) {
        this.price = price;
        System.out.println("Laptop: " + this.name());
    }

    // Getter for price
    public int getPrice() {
        return price;
    }

    // Setter for price
    public void setPrice(int price) {
        this.price = price;
    }
}

public class Demo {
    public static void main(String[] args) {
        Laptop lap = Laptop.Macbook;
        System.out.println("Selected Laptop: " + lap);

        // Display all enum constants with their prices
        for (Laptop l : Laptop.values()) {
            System.out.println(l + ": $" + l.getPrice());
        }
    }
}
```

Output:

```
Laptop: Macbook
Selected Laptop: Macbook
Macbook: $2000
XPS: $2200
Surface: $2200
ThinkPad: $1500
```

Explanation of Example

1. **Enum Constants with Prices:** In the Laptop enum, we define four constants: Macbook, XPS, Surface, and ThinkPad. Each of these constants is initialized with a price using the parameterized constructor.
2. **Private Variables and Constructor:** The enum has a private variable price, which stores the price for each constant. The constructor takes an int argument to initialize the price for each laptop.
3. **Getter and Setter Methods:** Since the price is private, we use a getter (getPrice()) to access the price value. We also provide a setter (setPrice()) to allow the modification of prices if necessary.
4. **Looping Through Enum Constants:** The values() method is used to loop through all the enum constants (Laptop.values()) and print each constant's name and its price.

Common Issues and Solutions

- **Omitting Values:** If you define a parameterized constructor for your enum but forget to initialize one of the constants with a value, the compiler will throw an error. To avoid this, either assign a value to every constant or provide a default constructor.
- **Constructor Visibility:** The constructor in an enum is always private. You do not need to explicitly specify the private keyword, but it is good practice to do so to make the visibility clear.

10.11-What is Annotation

Introduction:

Annotations in Java are a powerful feature introduced in Java 5. They act as a form of metadata, providing additional information to the compiler or runtime about the program elements (such as classes, methods, or variables). However, annotations do not alter the program's behaviour directly; instead, they can influence how the compiler processes the code.

- **What are Annotations?**
    - An annotation is essentially a tag or indicator attached to a class, interface, method, or field to provide supplementary data.
    - Annotations begin with the @ symbol.
    - They provide metadata that can be processed at compile-time or runtime by the compiler or the JVM.

Key Characteristics of Annotations:

- Annotations do not change the action of a compiled program.
- They help associate metadata (additional information) with program elements.
- Annotations can be applied to variables, methods, constructors, classes, etc.
- Unlike comments, annotations can be used by the compiler or runtime tools to change the way the program is processed.

Built-In Java Annotations

Java provides several built-in annotations that are widely used in coding. These annotations can either be applied directly to the code or to other annotations.

Common Built-In Annotations in Java:

1. **@Override**
   Indicates that a method overrides a method in the superclass. If the method does not actually override anything, a compile-time error will occur.

2. **@SuppressWarnings**
   Instructs the compiler to suppress specific warnings that it would otherwise generate. It's typically used to suppress warnings related to unchecked or deprecated operations.

3. **@Deprecated**
   Marks a method or class as outdated and discourages its use. When this annotation is present, the compiler issues a warning when the deprecated element is used.

Example: Understanding @Override Annotation

Here's a simple example that demonstrates the use of the @Override annotation:

```
class A {
    public void show() {
        System.out.println("In A's show");
    }
}

class B extends A {
    @Override
    public void show() {
        System.out.println("In B's show");
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new B();
        obj.show();
    }
}
```

Output:

```
In B's show
```

Explanation:

- In this example, class B overrides the show() method from class A. By adding the @Override annotation above the method, the compiler checks if the method correctly overrides a superclass method.

- If the method name is incorrect, the @Override annotation will raise a compile-time error, helping to prevent potential bugs. For example, if the method in B were mistakenly named showw(), the compiler would notify us that the method does not override anything from the parent class.

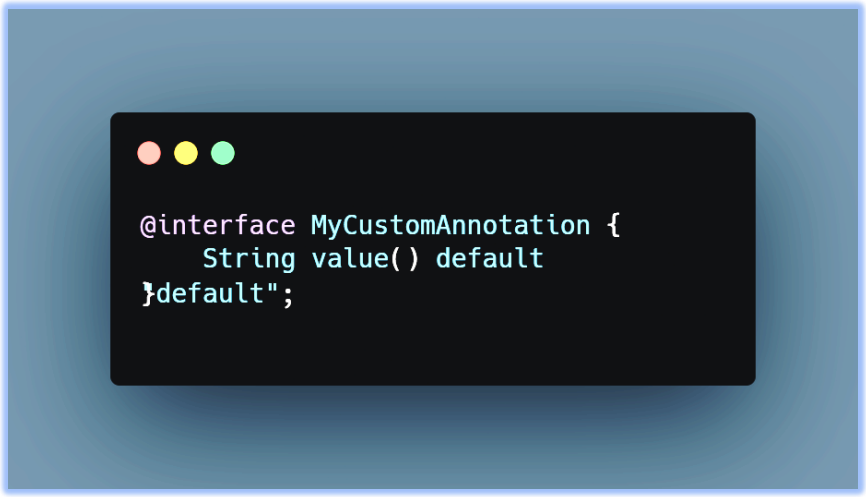
Why Use @Override?

- **Prevents Mistakes:** Helps detect errors when method names are mistyped or incorrectly defined.
- **Improves Code Readability:** Makes it clear to readers that the method is overriding a parent class method, which improves maintainability.

Custom Annotations in Java

Java also allows developers to create their own custom annotations. Custom annotations are declared using the @interface keyword. Although they are not commonly used in core Java, they become more relevant in advanced topics, particularly when dealing with frameworks like Spring or Hibernate.

Example of a Custom Annotation:



- The @interface keyword is used to define an annotation.
- The value element defines a default value for this annotation, which can be overridden when used.

Where to Use Annotations

Annotations can be applied at various levels within your code:

- **Class Level:** Can indicate that a class is deprecated or give other metadata about the class.
- **Method Level:** Used to show method overrides or suppress warnings.
- **Field/Variable Level:** Can mark fields as transient, inject dependencies, or associate metadata.

Annotation Usage in Java Frameworks

- While annotations may not be heavily used in basic Java programming, they are essential in Java frameworks. For example, annotations play a significant role in Spring, Hibernate, and other enterprise frameworks, helping automate tasks such as dependency injection, transaction management, and more.

10.15-Types of Interface

Interface in Java

In Java, an **interface** is a mechanism to achieve **abstraction**. Interfaces define a contract of methods that must be implemented by any class that implements the interface. Interfaces can only contain **abstract methods** (methods without a body) in earlier versions of Java, but from Java 8 onward, they can also contain **default** and **static methods**. Interfaces help to achieve multiple inheritance in Java, as a class can implement multiple interfaces.

Types of Interfaces in Java

There are three main types of interfaces in Java:

1. **Normal Interface**
2. **Functional Interface** (SAM – Single Abstract Method Interface)
3. **Marker Interface**

---

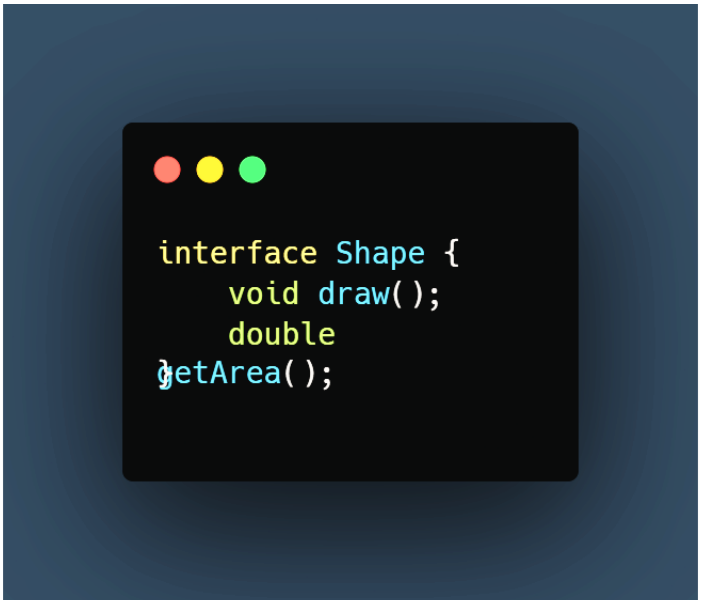
1. Normal Interface

A **normal interface** is an interface that contains **two or more abstract methods**. These interfaces are the most commonly used in Java. Any class that implements this interface must provide an implementation for all its abstract methods.

Starting from Java 8, normal interfaces can also include:

- **Default methods** (methods with a body that have a default implementation).
- **Static methods** (methods that belong to the interface class, not to an instance of the class).

Example:



In this example, Shape is a normal interface with two abstract methods: draw() and getArea(). Any class implementing this interface must define both methods.

---

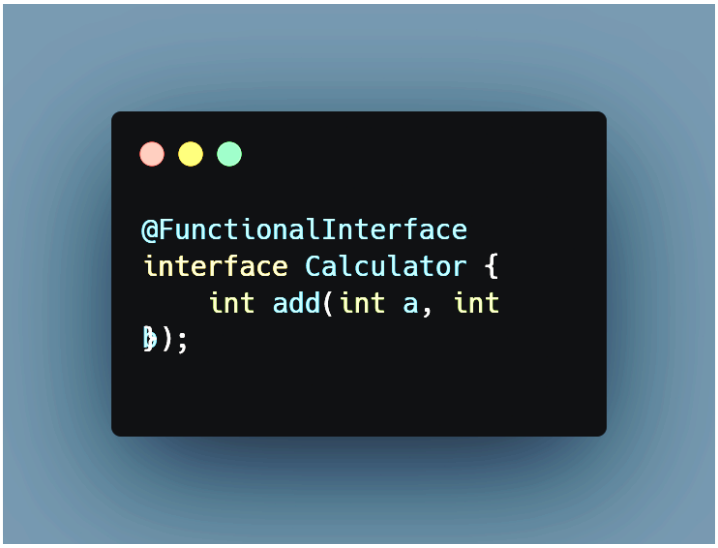
2. Functional Interface

A **functional interface** is an interface that contains **only one abstract method**. This type of interface is also called **SAM** (Single Abstract Method) interface. Functional interfaces can have any number of:

- **Static methods**
- **Default methods**
- **Methods from java.lang.Object**

Lambda expressions are typically used with functional interfaces in Java.

Example of Functional Interface:



Here, Calculator is a functional interface with a single abstract method add(int a, int b).

Important Note: Even though it has one abstract method, a functional interface can have multiple default and static methods.

---

3. Marker Interface

A **marker interface** is an interface that does **not contain any methods, fields, or constants**. Marker interfaces serve the purpose of **tagging** or **marking** a class as having a particular property. These interfaces rely on runtime behaviour.

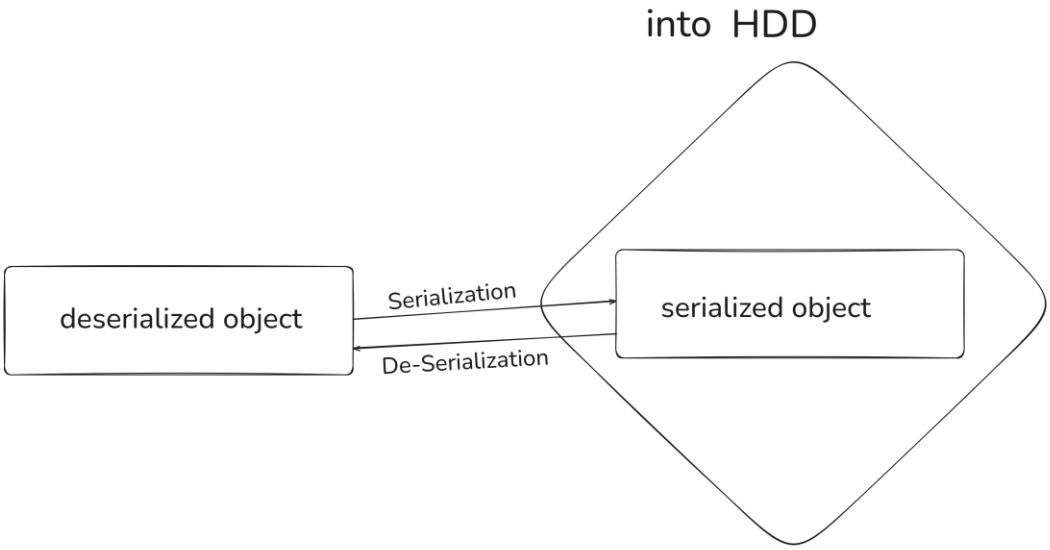
Examples of Marker Interfaces:

- **Serializable**
- **Cloneable**

Serializable Interface:

The **Serializable** interface is a marker interface defined in the java.io package. It is used to make a class **serializable**, meaning that the state of an object can be converted into a byte stream and stored in a file or database.

- **Serialization**: The process of converting an object's state to a byte stream, making it suitable for storage or transmission.
- **Deserialization**: The reverse process of restoring the object's state from a byte stream back into memory.



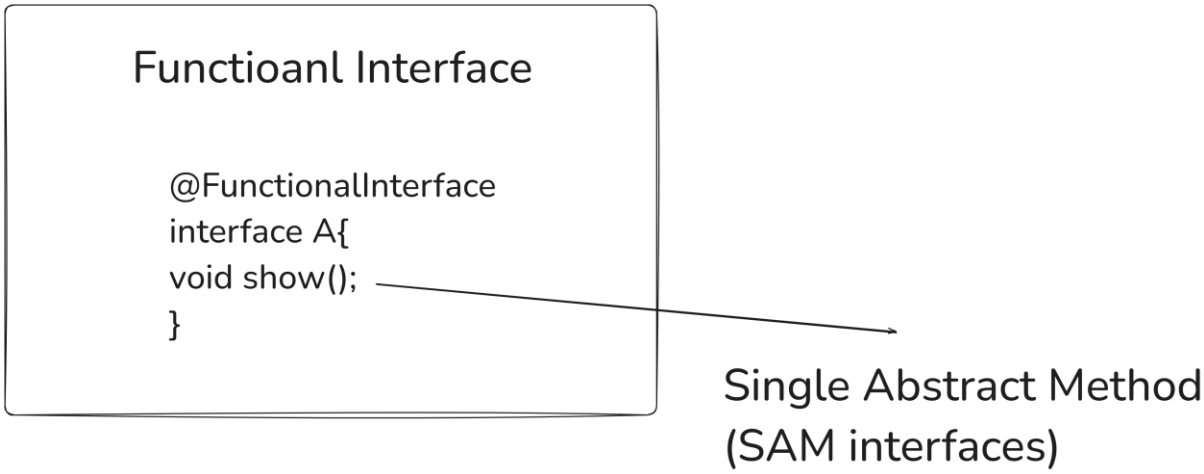
Key Points to Remember:

- **Normal Interface**: Contains two or more abstract methods. From Java 8 onward, they can also have default and static methods.
- **Functional Interface**: Contains only one abstract method. Can be used with lambda expressions.
- **Marker Interface**: Does not contain any methods or fields, used for tagging or marking classes (e.g., Serializable).

10.12-Functional Interface New

Introduction

A **functional interface** is an interface that contains only one abstract method. It is used when there is only one functionality that needs to be exhibited. In Java, functional interfaces are commonly referred to as **Single Abstract Method (SAM) Interfaces** because they have exactly one abstract method. These interfaces can have default and static methods, but only one abstract method. Functional interfaces are annotated with @FunctionalInterface to explicitly declare them as functional interfaces.



In Java SE 8 and later, functional interfaces work seamlessly with **lambda expressions** and **method references**, allowing cleaner, more readable code.

Example

```java
@FunctionalInterface
interface A {
    void show();
}

public class MyClass {
  public static void main(String args[]) {
            // Anonymous inner class instead of creating a separate class for implementation
        A obj = new A() {
            public void show() {
                System.out.println("in show");
            }
        };
        obj.show(); // Output: in show

  }
}
```

In this example, an anonymous inner class is used to provide an implementation for the show() method of interface A. Since A only has one method, it can be treated as a functional interface.

Built-In Java Functional Interfaces

From Java SE 1.8 onwards, several interfaces were converted into functional interfaces. These interfaces are annotated with @FunctionalInterface to enforce the functional interface design.

Here are some built-in functional interfaces:

- **Runnable**: Contains only the run() method.
- **Comparable**: Contains only the compareTo() method.
- **ActionListener**: Contains only the actionPerformed() method.
- **Callable**: Contains only the call() method.

These functional interfaces are widely used across Java applications and frameworks.

Functional Interfaces in Java 8

Java is often considered a verbose language because of its detailed and explicit code requirements. In Java 8, **lambda expressions** were introduced to reduce verbosity and make code cleaner and easier to understand. Lambda expressions are a key feature that works only with functional interfaces.

For example, instead of using anonymous inner classes, lambda expressions can simplify the code significantly.

Key Points

- A **functional interface** can only have one abstract method.
- Functional interfaces can have default and static methods, but only one abstract method.
- Lambda expressions and method references can be used with functional interfaces to simplify code.
- **@FunctionalInterface** annotation is used to explicitly declare an interface as functional, although it's optional.
- Functional interfaces are extensively used in **streams**, **concurrency**, and **event handling** in Java.

10.13-lamda Expression

Introduction to Lambda Expression

Lambda expressions provide a clear and concise way to represent the implementation of a **functional interface**. A functional interface is one that contains only a single abstract method, and with lambda expressions, we don't have to implement the method in a traditional way. Instead, we simply write the logic of the method, significantly reducing code.

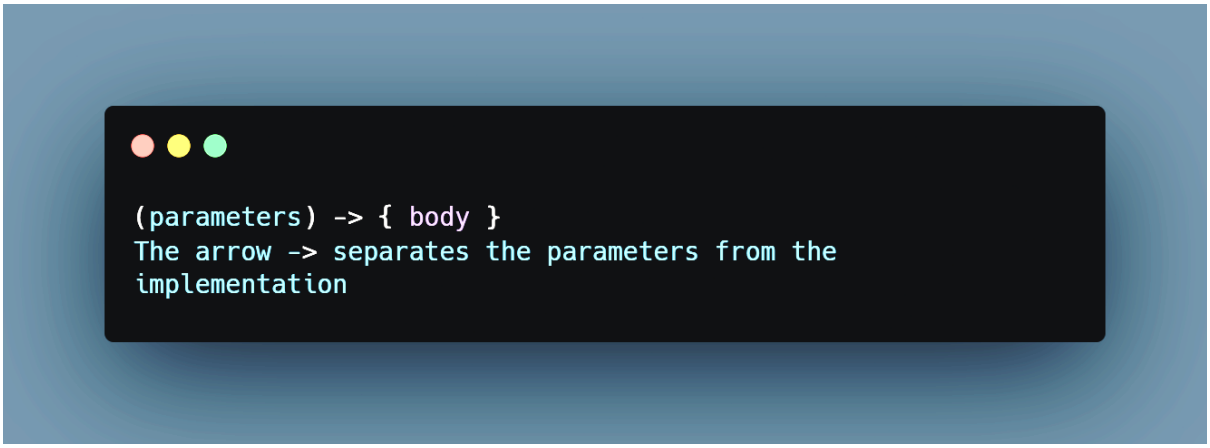
Lambda expressions were introduced in **Java SE 8** and are designed to make coding more efficient by reducing boilerplate code, especially when working with interfaces that have only one abstract method.

Functional Interface and Lambda Expression

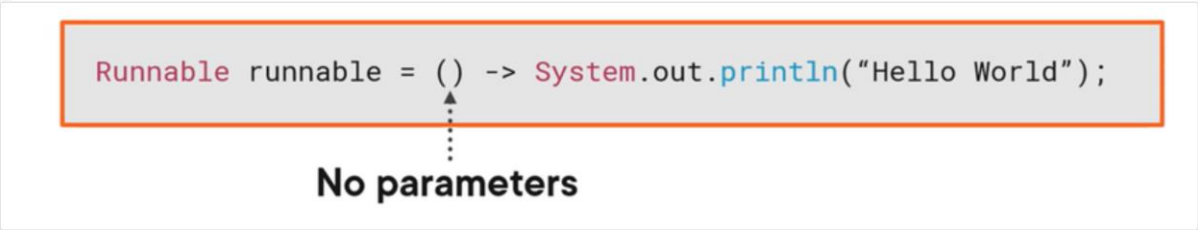
- A **functional interface** is an interface with a single abstract method, and lambda expressions allow us to provide a clean implementation of this method.
- The functional interface can be implemented using an anonymous inner class, but **lambda expressions** simplify this further.

Syntax of a Lambda Expression

Lambda expressions use the syntax:



Example of Lambda Expression with Zero Parameters:



Code example:

```
interface A {
    void show();
}

public class Demo {
    public static void main(String[] args)
{       A obj = () -> {
            System.out.println("in show");
        };
        obj.show();
    }
}
```

Output:

```
in show
```

Explanation:
In this example, we implemented the show() method of the A interface using a lambda expression. No need for an anonymous inner class or separate class for the implementation—just the logic within the lambda expression.

Why Use Lambda Expressions?

1. **Implementation of Functional Interface**: Lambda expressions provide a quick and readable way to implement functional interfaces.
2. **Less Coding**: They reduce the lines of code, especially for single-method interfaces.
3. **Syntactical Sugar**: This reduction in code and improved syntax readability is often referred to as "syntactical sugar," making code easier to read and maintain.

Lambda Expression with Parameters

When an interface method accepts parameters, lambda expressions can still be used to implement them easily.

Example of Lambda Expression with One Parameter:

```
interface A {
    void show(int i);
}

public class Demo {
    public static void main(String[] args) {
        A obj =  i -> System.out.println("in show: " +
i);     obj.show(5);
    }
}
```

Output:

```
in show: 5
```

Explanation:
In this example, we implemented the show(int i) method using a lambda expression that accepts an integer parameter i. Notice that the type of the parameter (int) is included before the parameter. If the method has only one parameter, we don't need parentheses around the parameter, and the type can also be omitted.

Simplified Lambda Expression With Two Parentheses:

```
interface A {
    void show(int i, int j);
}

public class Demo {
  public static void main(String args[]) {
    A obj = (i,j) -> System.out.println("in show: " + i +","+
j);     obj.show(5,6);

  }
}
```

Output:

```
in show: 5,6
```

Explanation:
Here, the two parameters type int and parentheses around the parameter i and j are necessary. This no parenthesis is allowed when there is only one parameter, making the code more concise.

Key Features of Lambda Expressions

- **No need to instantiate interfaces**: The compiler automatically handles the instantiation, making the code more readable.
- **Simplicity**: Instead of writing a full method implementation, the developer just focuses on the logic.
- **Efficiency**: Lambda expressions work efficiently with functional interfaces, reducing the need for boilerplate code.

FAQ's

What are Lambda Expressions in Java?

Lambda expressions are a short block of code that:

- Accept inputs as parameters.
- Return a result or perform an action without returning anything.

Is It Good to Use Lambda Expressions in Java?

Yes, lambda expressions improve code readability, reduce boilerplate, and make it easier to interact with functional interfaces, streams, and APIs introduced in Java 8 and later.

What Are the Drawbacks of Lambda Expressions?

The main limitation is that lambda expressions can only be used with functional interfaces (i.e., interfaces that have only one abstract method).

10.14-Lamda Expression with return type

Introduction to Lambda Expression with Return Type

A **lambda expression** can also return values. When using lambda expressions with methods that have a return type, the syntax can be optimized to reduce code significantly.

Below, we will explore how a method with two parameters and a return type is implemented first using an **anonymous inner class** and then optimized using a **lambda expression.**

Example: Anonymous Inner Class with Return Type

```
interface A {
    int add(int i, int j);
}
public class MyClass {
  public static void main(String args[]) {
    A obj = new A() {
            public int add(int i, int j) {
                return i + j;
            }
        };
        int result = obj.add(5, 4);
        System.out.println("The addition is: " +
result);
}
```

Output:

```
The addition is: 9
```

Explanation:

- In the above example, we are using an **anonymous inner class** to implement the add() method of the A interface.
- The add() method takes two integer parameters (i and j) and returns their sum.
- This approach works, but the code is somewhat **verbose.**

Optimized Example: Lambda Expression with Return Type

Now, we will optimize the same implementation using a lambda expression.

```
interface A {
    int add(int i, int j);
}
public class MyClass {
  public static void main(String args[]) {
   A obj = (i, j) -> i + j;
        int result = obj.add(5, 4);
        System.out.println("The addition is: " +
result);
}
```

Output:

```
The addition is: 9
```

Explanation:

- The **lambda expression** (i, j) -> i + j directly implements the add() method.
- No need for the return keyword, as the expression itself acts as the return value.

- We don't need to specify the data type of the parameters (i and j) because they are inferred automatically.
- This version of the code is significantly shorter and more readable compared to the anonymous inner class implementation.

Key Points to Remember About Lambda Expressions with Return Type:

1. **Return Type Omission**:
    - If the lambda body contains a single return expression, the return keyword can be omitted. The result of the expression is automatically returned.
2. **Parameter Type Omission**:
    - If the method parameters have the same type, we can omit the parameter type in the lambda expression. For example, both i and j are integers, so their types are inferred automatically.

Usage of Lambda Expressions in Collections

Lambda expressions can also be used as parameters when working with collections. They are particularly useful in functional operations like **filtering**, **mapping**, and **sorting**.

Important Points to Remember:

- **Lambda expressions** can only be used with **functional interfaces**. A functional interface has exactly one abstract method. If an interface has more than one method, using a lambda expression would lead to ambiguity, as it wouldn't be clear which method the lambda is implementing.
- Lambda expressions **simplify code** by reducing the need for boilerplate code, especially when implementing interfaces with single methods.
- **Type Inference**: Java automatically infers parameter types in lambda expressions, making the code more concise.

Additional Points to Consider:

- **Code Readability**: While lambda expressions reduce the amount of code, it's essential to use them wisely. Overusing lambda expressions in complex logic might make the code harder to understand.
- **Functional Programming**: Lambda expressions are part of the broader functional programming capabilities introduced in Java 8. They work seamlessly with **streams**, **functional interfaces**, and **method references**.

11.1-What is an Exception

What is an Exception in Java?

An **exception** in Java is an event that disrupts the normal flow of a program. Exceptions occur during the execution of a program (runtime) and can cause the program to terminate unexpectedly unless properly handled. Exception handling allows a program to handle errors gracefully and continue its execution.

Errors in Java

In Java, an **Error** signifies a serious issue, indicating that a normal Java application should not attempt to handle it. Errors are often associated with abnormal conditions, such as hardware failures or critical system-level issues, which are beyond the control of the program.

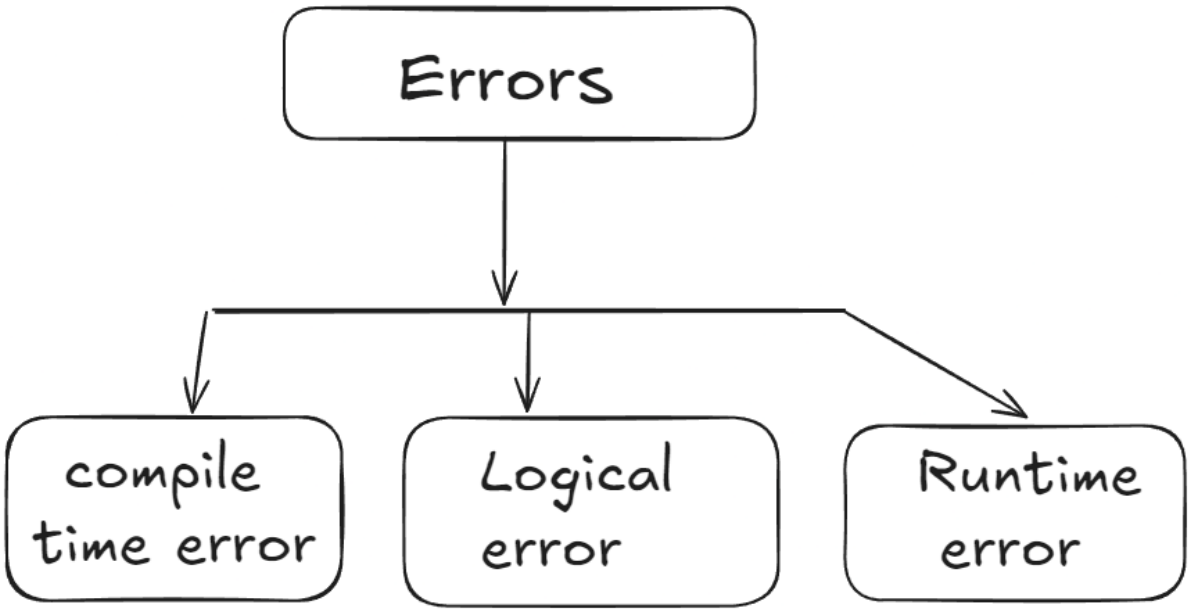
Unlike exceptions, errors cannot be resolved or handled through normal programming techniques. When an error occurs, it typically causes the program to terminate because errors represent conditions that the application should not try to recover from.

Types of Errors

In programming, there are three major types of errors that developers encounter:

1. **Compile-time errors**
2. **Logical errors**

3. **Runtime errors (exceptions)**



---

1. Compile-time Errors

Compile-time errors are errors that are detected by the compiler when a program is being compiled. These errors occur due to incorrect syntax, missing files, or other issues that prevent the program from being compiled.

Example:

```
public class Demo {
    public static void main(String[] args)
{       System.out.prinltn("Hello World");
    }
}
```

Error: In this code, there is a spelling mistake in System.out.println, where prinltn should be println.

Explanation:

- The error is detected during compilation because of incorrect syntax (misspelled method name).
- After fixing the error, the program will compile and run successfully.

---

2. Logical Errors

Logical errors are errors that occur when the program runs successfully, but the output is not what was expected. These errors do not cause compilation or runtime issues, but they cause incorrect results.

Example:

```
public class Calculator {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        System.out.println("Sum: " + (a - b)); // Incorrect
logic
}
```

Error: Here, the intention is to add two numbers (a + b), but instead, the code performs a subtraction (a - b), which results in incorrect output.

Explanation:

- Logical errors occur due to a flaw in the algorithm or logic of the program.
- These errors are harder to detect because they don't stop the execution but produce incorrect results.
- To fix these errors, testing and debugging are required to identify the problem in the logic.

---

3. Runtime Errors (Exceptions)

Runtime errors occur while the program is running. These errors typically happen when something unexpected occurs, such as trying to divide by zero, accessing an invalid index in an array, or working with unavailable resources (like files). These errors are also called **exceptions**.

Example: Suppose you write a program to open and read a file from your system:

```
import java.io.*;
public class FileReaderDemo {
    public static void main(String[] args) throws IOException
{       FileReader fr = new FileReader("nonexistentfile.txt");
        BufferedReader br = new BufferedReader(fr);
        System.out.println(br.readLine());
    }
}
```

Error:

- If the file nonexistentfile.txt does not exist, the program will throw a **FileNotFoundException** at runtime, causing the program to terminate abruptly.

Explanation:

- Runtime errors (exceptions) occur after the program has compiled successfully but face an issue while running.
- Such errors could be critical if they happen in a real-world application (e.g., trying to access an important or confidential file).
- **Exception Handling** is the mechanism that allows the program to handle these errors gracefully without crashing.

Handling Runtime Errors (Exception Handling)

In real-world applications, it's crucial that runtime errors are managed appropriately. For instance, if a file is missing, the program should not crash but instead notify the user and continue running.

This process of anticipating, detecting, and managing errors is known as **Exception Handling**. By using **try-catch** blocks, we can manage exceptions in a structured way, ensuring that the program doesn't abruptly terminate.

Example of Handling an Exception:

```
import java.io.*;

public class FileReaderDemo {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("nonexistentfile.txt");
            BufferedReader br = new BufferedReader(fr);
            System.out.println(br.readLine());
        } catch (FileNotFoundException e) {
            System.out.println("File not found. Please check the file
path.");} catch (IOException e) {
            System.out.println("An I/O error occurred.");
        }
    }
}
```

Explanation:

- In this code, we handle the potential FileNotFoundException using a try-catch block.
- Instead of the program terminating abruptly, the error message is displayed, and the program continues.

Here's a refined version of your notes on **Errors in Java** with improvements in structure, grammar, and clarity. I've also included suggestions on where to add images and provided feedback at the end.

Errors in Java

In Java, an **Error** is a subclass of **Throwable** that signifies a serious issue, indicating that a normal Java application should not attempt to handle it. Errors are often associated with abnormal conditions, such as hardware failures or critical system-level issues, which are beyond the control of the program.

Unlike exceptions, errors cannot be resolved or handled through normal programming techniques. When an error occurs, it typically causes the program

to terminate because errors represent conditions that the application should not try to recover from.

Key Characteristics of Errors

- **Non-Recoverable Issues**: Errors are usually related to the environment in which the application is running, such as system crashes, memory leaks, or hardware failures.
- **Program Termination**: When an error occurs, it usually results in the termination of the program. Unlike exceptions, errors cannot be caught or resolved using try-catch blocks.
- **Cannot Be Handled**: Since errors indicate serious problems, Java does not expect developers to handle them. They are beyond the scope of normal recovery strategies.

Summary: Key Points on Exceptions

- **Compile-time errors** are detected by the compiler and usually result from syntax issues.
- **Logical errors** occur when the program runs but produces incorrect results due to flawed logic.
- **Runtime errors (exceptions)** are unexpected issues that arise when the program is running, such as accessing unavailable resources.

By using **Exception Handling**, runtime errors can be managed, ensuring that the program continues to function smoothly and avoids abrupt termination.

11.2-Exception handling using try catch

In Java, when we run programs, we expect them to compile and execute smoothly without encountering any problems or exceptions. However, that's not always the case. To ensure our programs handle potential issues, we need a mechanism to manage these situations. This is where **exception handling** comes in.

Types of Statements in a Program

In a Java program, we deal with two types of statements:

1. **Critical Statements**
2. **Normal Statements**

1. Critical Statements (Risky Code)

- These are the blocks or lines of code that may potentially cause exceptions.
- We refer to these statements as ***risky code*** because they are prone to fail under certain conditions (e.g., dividing by zero, accessing a null object).

2. Normal Statements

- These are the parts of the program that execute successfully without any issues.
- These statements compile and run fine without triggering any exceptions.

Real-Life Example

Think of a scenario where you're holding the hand of a child in a park while the child is walking & playing around. This is a **normal situation** because it's a controlled environment with no real danger.
However, when you're walking on a busy road, and the child tries to run towards the traffic, this becomes a **critical situation**. You need to hold the child's hand tightly to prevent any accidents.

In programming terms, handling risky code (similar to walking on the road) is essential to avoid unexpected errors that might cause the program to stop.

Example of Exception in Java

Let's look at a simple example that demonstrates an exception.

```java
public class Demo {
    public static void main(String[] args) {
        int i = 0;  // Changed value of i to 0, which will cause an error
        int j = 18 / i;  // Risky statement, causes an ArithmeticException
        System.out.println(j);  // Unreachable code
    }
}
```

Output:

```
An ArithmeticException occurs with the message: / by 0
```

.

When an exception occurs (in this case, dividing by zero), the program's execution halts at that point, and the subsequent lines of code become **unreachable**. This is why handling exceptions is crucial for smooth program flow.

Handling Exceptions

To handle exceptions in Java, we use a mechanism that separates risky code into a special block. This allows us to handle potential errors without stopping the program.

1. The Try Block

- A **try block** is used to wrap the risky or critical code.
- If an exception occurs within this block, it is caught and handled by the corresponding **catch block**.
- If no exception occurs, the program continues executing normally, skipping the catch block.

2. The Catch Block

- A **catch block** is used to handle the exceptions thrown by the try block.
- The catch block contains code that will execute if an exception occurs.

- This block accepts an object of the Exception class, allowing us to customize the message or actions taken when an exception is caught.

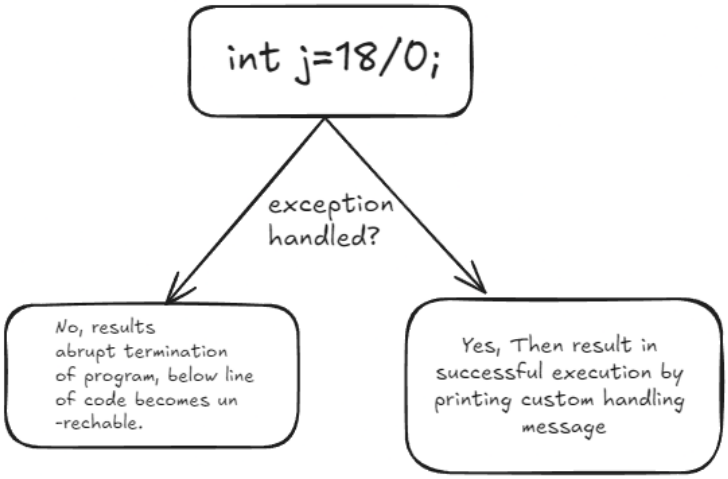
Example of Try-Catch Block in Java

```
public class Demo {
    public static void main(String[] args) {
        try {
            int i = 0;
            int j = 18 / i;  // This will throw an exception
            System.out.println(j);  // This below lines won't execute if an exception occurs
            System.out.println  ("Bye");
        } catch (ArithmeticException e) {
            System.out.println("something went wrong....");  // Custom message for handling exception
        }
    }
}
```

Output:

```
Bye
Something went wrong
```

How Try-Catch Works:



- If the code in the try block executes without any issues, the catch block is **skipped**.
- If an exception is encountered, the control jumps to the catch block, which handles the exception, preventing the program from crashing.

11.3-try with multiple catch

In Java, exceptions are used to handle runtime errors. Often, we may need to handle multiple types of exceptions that could occur within a program. Java provides a mechanism called "**multiple catch blocks**" to handle different exceptions separately and appropriately.

Exploring Exceptions

Example 1: Handling a Single Exception

```java
public class Demo {
    public static void main(String[] args) {
        int i = 0, j = 0;

        try {
            j = 18 / i;  // Risky code - may throw ArithmeticException
        } catch (Exception e) {
            System.out.println("Something went wrong: " + e);  // Prints exception message
        }

        System.out.println(j);  // Will not be executed if exception occurs before
        System.out.println("Bye");
    }
}
```

Output:

```
Something went wrong: java.lang.ArithmeticException: / by zero
Bye
```

Explanation:

- In this example, we attempt to divide by zero, which triggers an ArithmeticException.
- The exception is caught in the catch block, and the message is displayed: "Something went wrong: / by zero".
- The program continues and prints "Bye", preventing abrupt termination due to the error.

Key Concept:

- The exception object (e) in the catch block provides useful information about the specific error, making debugging easier. Instead of terminating the program, we handle the error gracefully and allow subsequent lines to execute.

Modifying the Code to Add More Exceptions

Now, let's add a few more risky operations to demonstrate how different exceptions can occur.

Example 2: Handling Another Type of Exception

```java
public class Demo {
    public static void main(String[] args) {
        int i = 2, j = 0;

        try {
            int[] nums = new int[5];
            System.out.println(nums[5]);  // ArrayIndexOutOfBoundsException
            j = 18 / i;  // Risky code - may throw ArithmeticException
        } catch (Exception e) {
            System.out.println("Something went wrong: " + e);
        }

        System.out.println(j);  // Still gets executed
        System.out.println("Bye");
    }
}
```

Output:

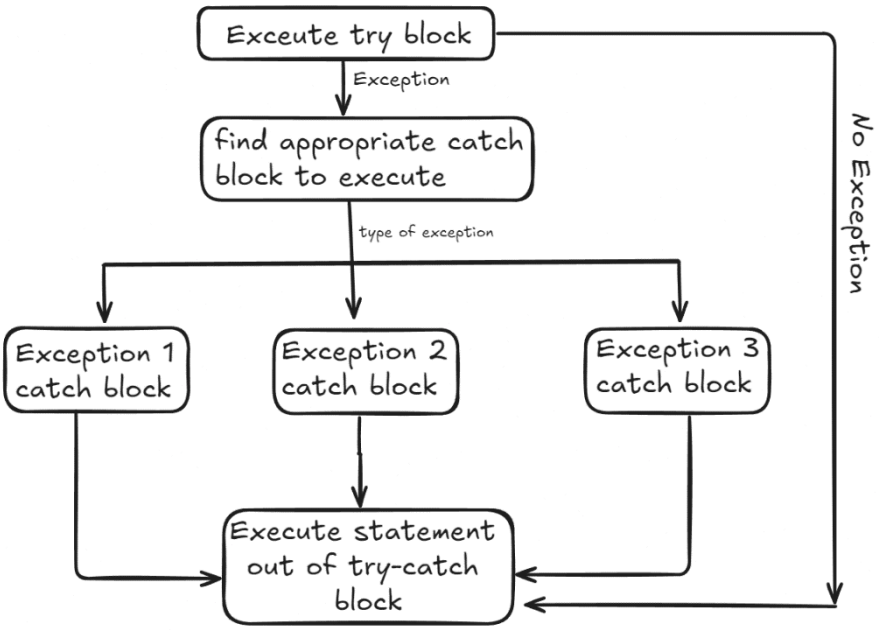
```
Something went wrong: java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
Bye
```

Explanation:

- We attempted to access an array element beyond its valid range, causing an ArrayIndexOutOfBoundsException.
- This error is caught by the catch block, which prints the error message, and the program continues to execute normally.

Try with Multiple Catch Blocks

When multiple types of exceptions may occur in a try block, we can handle them using **multiple catch blocks**. This way, we can provide specific handling for each exception type.



Example 3: Handling Multiple Exceptions Separately

```
public class MyClass {
  public static void main(String args[]) {

        int i = 2, j = 0;
        try {
            String str = null;
            System.out.println(str.length());   // Risky code - may throw Exception

            j = 18 / i;
            int[] nums = new int[5];
            System.out.println(nums[1]);  // Safe, no exception here
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds.");
        } catch (Exception e) {
            System.out.println("Something went wrong: " + e);
        }
        System.out.println(j);
        System.out.println("Bye");
  }
}
```

Output:

```
Something went wrong: java.lang.NullPointerException: Cannot invoke "String.length()"
0
Bye
```

Explanation:

- In this example, we have multiple catch blocks, and a new kind of Exception i.e **NullPointerException** is emerged which the Generic Exception catah block has caught it .Here, each catchdesigned to handle a specific type of exception:
    - **NullPointerException**: Occurs when trying to access the properties of a null object (str.length()).
    - **ArithmeticException**: Handles division by zero.
    - **ArrayIndexOutOfBoundsException**: Handles invalid array indexing.
    - **Exception**: This is the parent class for all exceptions. It catches any remaining exceptions that aren't handled by the previous catch blocks.

Key Takeaway:

- Specific catch blocks handle different exceptions, providing appropriate error messages.
- A final generic Exception catch block handles any other exceptions that aren't anticipated, ensuring the program remains robust.

Why Use Multiple Catch Blocks?

Handling exceptions with multiple catch blocks offers several advantages:

- **Specific Handling:** Each exception can be dealt with individually, allowing for more specific and meaningful error messages.
- **Improved Debugging:** By knowing exactly which type of error occurred, debugging becomes easier.
- **Code Continuation:** Despite encountering an exception, the program can continue to execute following the catch block, preventing abrupt termination.

Common Exceptions in Java

Here are a few of the most common exceptions that developers handle using try-catch blocks:

- **ArithmeticException:** This exception is thrown when an exceptional arithmetic condition occurs, such as division by zero.
- **ArrayIndexOutOfBoundsException:** This is thrown when an array is accessed with an illegal index. The index might be negative or greater than or equal to the array's size.
- **NullPointerException:** This is raised when an attempt is made to access an object or method on a null reference.

Exception Hierarchy in Java

Java exceptions follow a hierarchy. Understanding this structure helps in catching specific exceptions and writing cleaner code.

- **Throwable**
    - **Exception**
        - **RuntimeException**
            - **ArithmeticException**
            - **ArrayIndexOutOfBoundsException**
            - **NullPointerException**

Key Points:

- **Exception** is the parent class for all exceptions.
- **RuntimeException** and its subclasses are unchecked exceptions, meaning they don't need to be explicitly caught or declared.
- **Throwable** is the top-level parent for both exceptions and errors.

11.4- Exception Hierarchy

Exception Hierarchy in Java

Introduction:

In Java, all classes inherit from the topmost superclass, the Object class. This class is implicitly extended by all other classes, and the same applies to exception handling. Throwable class is the root class of Java Exception hierarchy and immediate parent of all exceptions.

Note: Throwable is the only class in Java whose name ends with "able" and is a class, whereas other terms ending with "able" like Runnable, Callable, and Serializable are interfaces.

Throwable Class Hierarchy:

The Throwable class has two main subclasses:

1. **Error**
2. **Exception**

---

1. Error

Errors are severe problems that arise during program execution, which generally cannot be resolved. When an error occurs, it usually leads to program termination. Errors are not meant to be handled in a program.

Examples of Errors:

- **Thread Death**: Occurs when a thread (the smallest unit of a program) dies unexpectedly.
- **I/O Error**: Issues related to input/output operations.
- **Virtual Memory Error**: Occurs due to insufficient memory. This can also lead to an **OutOfMemoryError**, which is an internal problem that can't be handled within the program.

---

2. Exception

Exceptions are events that can be caught and handled by the program. They occur due to unexpected or abrupt execution of the program. Once an exception occurs, any lines of code following it are not executed unless the exception is properly handled.

Types of Exceptions:

Exceptions are categorized into two main types:

1. **Checked Exceptions**
2. **Unchecked Exceptions (Runtime Exceptions)**

Checked Exceptions:

These exceptions are known to the compiler and must be handled; otherwise, they will result in a compile-time error. You can either handle them using try-catch blocks or "duck" them by using the throws keyword.

- **Example Checked Exceptions**:
  - IOException
  - SQLException
  - ClassNotFoundException

Key Point: Checked exceptions are called "**compile-time exceptions**" because the compiler forces you to handle them.

Unchecked Exceptions (Runtime Exceptions):

These exceptions occur at runtime, after the program has successfully compiled. If an unchecked exception occurs and it is not properly handled, the program may crash.

- **Example Unchecked Exceptions**:
  - ArithmeticException
  - NullPointerException
  - ArrayIndexOutOfBoundsException

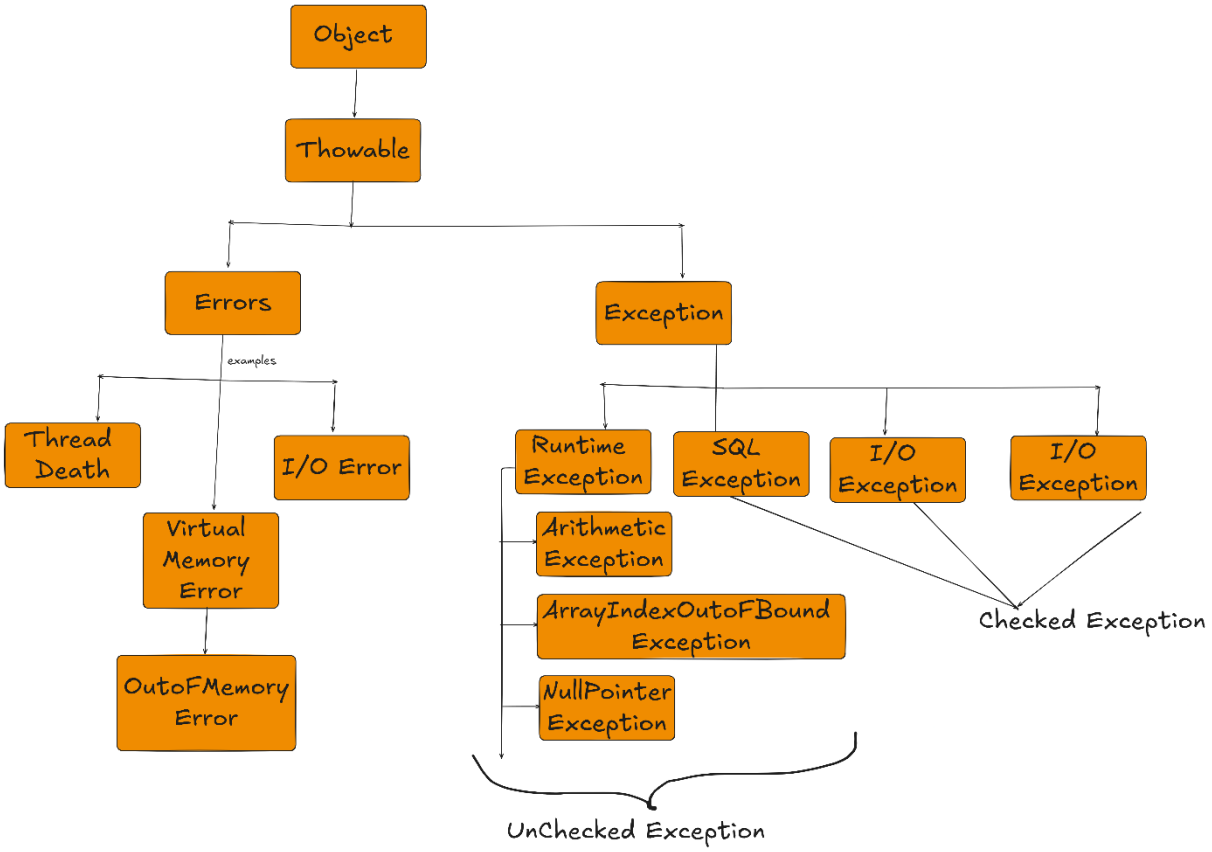
Unchecked exceptions do not need to be declared or handled at compile-time. The program will not show any errors at compile-time even if they are not handled.

Runtime Exceptions:

Runtime exceptions occur after the compilation of the program. If not handled by the program, they are caught by the default handler (JVM exception handler). However, it's a bad practice to rely on the default handler, and you should handle these exceptions in your code.

Common Runtime Exceptions:

- **ArithmeticException:** Occurs when there is a mathematical error, like dividing by zero.
- **ArrayIndexOutOfBoundsException:** Occurs when you try to access an index outside the bounds of an array.
- **NullPointerException:** Occurs when you try to use a null object reference.



Differences Between Checked and Unchecked Exceptions:

Aspect	Checked Exceptions	Unchecked Exceptions
Known at Compile-Time	Yes, must be handled by the programmer.	No, known only at runtime.
Handling	Must be handled or declared using throws.	Can be handled optionally; no compile error.
Examples	IOException, SQLException, ClassNotFoundException	NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException
Category	Known as compile-time exceptions.	Known as runtime exceptions.

TELUSKO

11.5-Exception throw keyword

Introduction to throw Keyword

In Java, when an exception occurs, it is typically handled using **try-catch** blocks. However, we can also manually generate exceptions using the **throw** keyword. The throw keyword is used to explicitly throw an exception during the execution of a program.

This is helpful when you want to create and manage your own custom exceptions or handle specific error conditions in a controlled manner. For example, you might throw an exception when user input is invalid, when a server fails to respond, or when certain logical conditions in your code are met.

- **Syntax of throw keyword**:

```
throw new ExceptionClass("error message");
```

Here, an instance of the exception class is created with an error message, and the throw keyword passes this exception to the appropriate catch block.

When to Use the throw Keyword

You can use the throw keyword to throw both **checked** and **unchecked** exceptions. This is useful when:

- You want to define your own conditions and throw an exception explicitly.
- You want to generate an exception based on specific logic (e.g., dividing by zero or invalid data).

For example, if you're dividing two numbers, you can throw an ArithmeticException if the denominator is zero.

Example 1: Handling ArithmeticException

```
public class ThrowExample {
    public static void main(String[] args) {
        int i = 0, j = 0;

        try {
            j = 18 / i; // Division by zero
        } catch (ArithmeticException e) {
            j = 18 / 1; // Fallback value to avoid zero division
            System.out.println("Caught an ArithmeticException, setting
default value: " + j);
        }

        System.out.println("Final value of j: " + j);
        System.out.println("End of Program");
    }
}
```

Explanation:

- In the above code, i is set to 0. When the program tries to divide by zero, an **ArithmeticException** is thrown.
- The catch block catches the exception and sets a default value of 18.
- The program continues to execute after handling the exception.

Example 2: Manually Throwing an Exception

```
public class ManualThrowExample {
    public static void main(String[] args) {
        int i = 20, j = 0;

        try {
            j = 18 / i; // Division operation
            if (j == 0) {
                throw new ArithmeticException("Manually thrown exception:
Division by zero not allowed.");
            }
        } catch (ArithmeticException e) {
            j = 18 / 1; // Default value
            System.out.println("Caught ArithmeticException, setting
default value: " + j);
        }

        System.out.println("Final value of j: " + j);
        System.out.println("End of Program");
    }
}
```

Explanation:

- In this example, a condition is added that manually throws an **ArithmeticException** when j equals 0.
- Even though no actual division by zero occurs in the program, the exception is thrown explicitly, allowing custom error handling.

This technique is commonly used in scenarios like database connections, where **fallback strategies** are needed when a primary resource (like a database) fails.

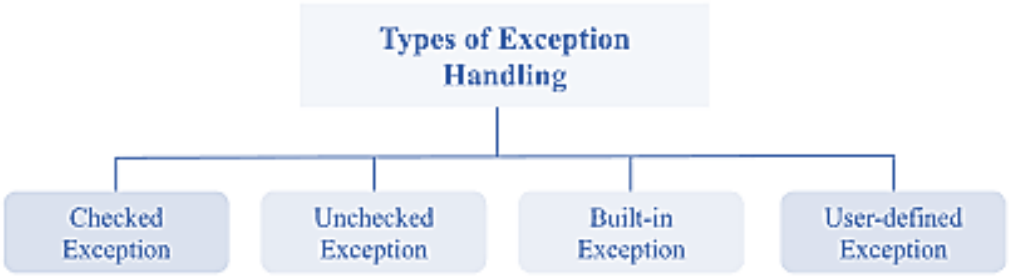
Key Points About the throw Keyword:

- **Use it for Custom Exceptions**: You can create and throw custom exceptions based on your application's needs.
- **Explicit Handling**: It allows explicit error handling at specific points in your code, offering more control over how and when exceptions are triggered.
- **Default Fallbacks**: You can throw exceptions and handle them gracefully by providing fallback solutions or retry mechanisms, ensuring smooth execution flow.

11.6-Custom Exception

Introduction to Custom Exceptions in Java

In Java, the throw keyword is often used to create **custom exceptions**, in addition to handling built-in exceptions (whether checked or unchecked). A **custom exception** (also known as a **user-defined exception**) is a class that extends Java's Exception class. Custom exceptions allow developers to define error conditions that are specific to their application's logic.



Java has a rich set of built-in exceptions that cover most error situations, but there are cases where you might want to throw a more **descriptive**, **application-specific exception.** In such cases, custom exceptions come into play.

Why Use Custom Exceptions?

Even though Java's built-in exceptions cover many error scenarios, there are times when custom exceptions are useful. Below are a few reasons to use custom exceptions:

- **Business Logic Exceptions**: Custom exceptions are useful for representing errors that are unique to your application's business logic, making them clearer and more descriptive.
- **Better Debugging**: By defining custom exceptions, developers and users can better understand the exact problem that occurred, which makes debugging easier.
- **Categorizing Errors**: Custom exceptions allow you to categorize certain errors and provide specific handling for them.

How to Create Custom Exceptions in Java

Creating a custom exception involves extending the Exception class (or RuntimeException if it is an unchecked exception). A custom exception can

have its own fields, methods, and constructors. The most common use is to provide a custom error message.

Here's how to create a custom exception in Java:

Example:

```
// Custom Exception Class
class NavinException extends Exception {
    public NavinException(String message) {
        super(message); // Passes the error message to the parent Exception class
    }
}

// Main Class
public class Hello {
    public static void main(String[] args) {
        int i = 20, j = 0;

        try {
            j = 18 / i; // Division operation
            if (j == 0) {
                throw new NavinException("Manually thrown exception: Division by zero
not allowed.");
            }
        } catch (NavinException e) {
            j = 18 / 1; // Provide a default value
            System.out.println("Custom Exception Caught: " + e);
        }

        System.out.println("Final value of j: " + j);
        System.out.println("End of Program.. Bye!");
    }
}
```

Output:

```
Custom Exception Caught: NavinException: Manually thrown exception: Division by zero
not allowed.
Final value of j: 18
End of Program.. Bye!
```

Explanation:

1. **Creating the Custom Exception:**
    - We created a class **NavinException** that extends the **Exception class.**
    - The constructor of NavinException accepts a string message, which is passed to the parent Exception class using the **super() method.** This enables the custom exception to display an error message.

2. **Throwing the Exception:**
    - Inside the try block, we perform a division operation (18 / i). If the value of j is zero (indicating a division by zero), we manually throw a new NavinException with a custom error message.
3. **Catching the Exception:**
    - In the catch block, the custom exception is caught, and we provide a default value for j (18). The message from the exception is printed along with the default value.
4. **End of Program:**
    - The program prints the final value of j and a "goodbye" message.

Things to Note:

- The super() method is used to call the parent constructor (Exception) and pass the custom error message to it.
- **Custom exceptions** can be used for specialized error conditions that are not covered by Java's built-in exceptions.
- You can also extend the **RuntimeException** class if you want the custom exception to be **unchecked**.

11.7-Ducking Exception using throws

Introduction:

The Java **throws** keyword is used to declare an exception in a method's signature. It informs the programmer that an exception **may** occur during the execution of the method. As a result, it encourages programmers to provide appropriate exception-handling code to maintain the normal flow of the program.

Key Points:

- The throws keyword is mainly used to handle **checked exceptions**.
- If an unchecked exception (e.g., NullPointerException) occurs, it is typically considered the programmer's fault for not verifying the code before execution.

Syntax of the throws Keyword:

```
return_type method_name() throws exception_class_name {
    // method code
}
```

Advantages of Using the throws Keyword:

1. **Propagation of Checked Exceptions:** Checked exceptions can be forwarded up the call stack.
2. **Provides Information to the Caller:** The caller of the method is informed about possible exceptions, allowing them to handle it properly.

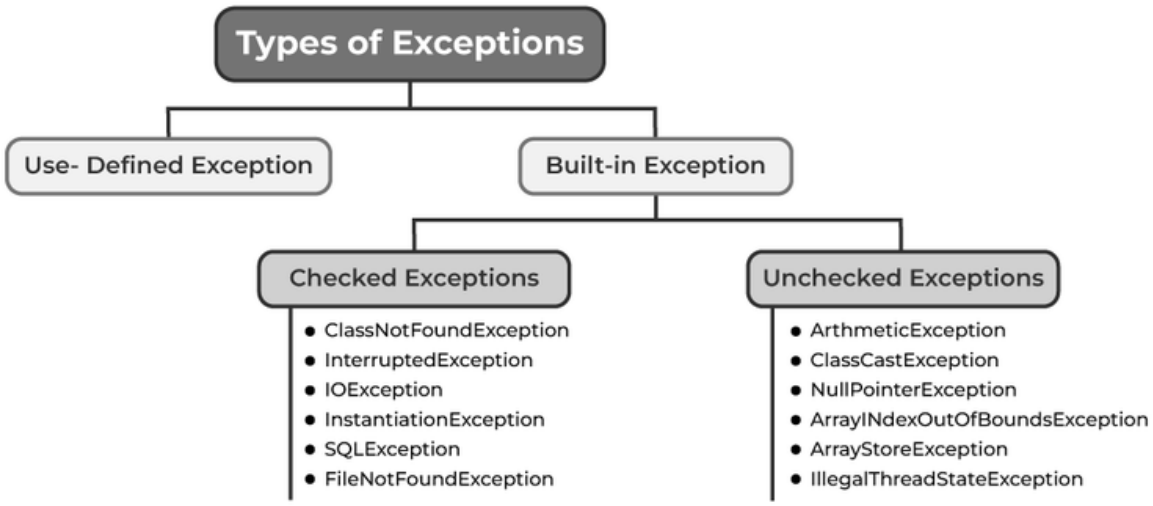
Scenario: Ducking Exceptions

Consider a method **d()** that contains multiple statements, one of which is **critical**. A critical statement is one that might throw an exception. Additionally, two other methods, **e()** and **f()**, each also have a critical statement.

Now, let's assume the main() method calls all of these methods. Since each method contains critical code that might throw an exception, there are two ways to handle this situation:

- **Handle the exception** within each method using a try-catch block.
- **Duck the exception** by not handling it within the method but rather passing it on to the method's caller using the **throws keyword.**

In our case, the main() method will call these critical methods. Each method (e.g., d(), e(), and f()) throws the exception back to the main() method instead of handling it. This is achieved using the throws keyword followed by the exact exception type (or its parent class).



At this point, it is up to the main() method to either handle the exception or duck it further. However, it's considered bad practice to

let the main() method throw the exception without handling it. If the main() method also ducks the exception, the Java Virtual Machine (JVM) will handle it using its **default exception handler**, which is not ideal in real-world coding.

Example:

```
class A {
    public void show() throws ClassNotFoundException {
        System.out.println("In show");
        Class.forName("Calculator"); // Might throw ClassNotFoundException
    }
}

public class Hello {
    public static void main(String[] args) {
        A obj = new A();
        try {
            obj.show();
        } catch (ClassNotFoundException e) {
            System.out.println("Class not found: " + e);
            e.printStackTrace();
        }
    }
}
```

Output:

```
In show
Class not found: java.lang.ClassNotFoundException: Calculator
java.lang.ClassNotFoundException: Calculator
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoade
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:526)
    at java.base/java.lang.Class.forName0(Native Method)
    at java.base/java.lang.Class.forName(Class.java:421)
    at java.base/java.lang.Class.forName(Class.java:412)
    at A.show(MyClass.java:17)
    at MyClass.main(MyClass.java:24)
```

Explanation:

In this example, the A class contains a **show()** method, which calls **Class.forName().** This method might throw a **ClassNotFoundException** (a checked exception). Instead of handling

the exception within show(), it throws it to the main() method using the throws keyword.

The main() method surrounds the show() call with a try-catch block, ensuring the exception is caught and handled properly.

The **e.printStackTrace()** method outputs the stack trace, which shows the method call hierarchy, tracing the path of execution from the origin of the exception to the point where it occurred.

Differences Between throw and throws

Sr. No.	Basis of Differences	throw	throws
1	**Definition**	Used to explicitly throw an exception inside a block of code.	Used in the method signature to declare exceptions that might be thrown during execution.
2	**Usage**	Can propagate unchecked exceptions only.	Can declare both checked and unchecked exceptions but is primarily used for checked exceptions.
3	**Syntax**	Followed by an instance of the exception to be thrown.	Followed by the class names of the exceptions.
4	**Declaration**	Used inside the method body.	Used with the method signature.

Sr. No.	Basis of Differences	throw	throws
5	**Internal Implementation**	Only one exception can be thrown at a time.	Multiple exceptions can be declared using throws. For example: main() throws IOException, SQLException.

11.8-User Input using BufferedReader and Scanner

Introduction:

In Java, when we want to print output to the console, we use the statement:

```
System.out.println();
```

In this statement, **println()** is a method from the **PrintStream** class, while out is a static and final object of the **System** class.

Breaking it down:

- System is the class.
- out is the static object.
- println is the method from the PrintStream class.

Example:

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

Output:

```
Hello
```

Explanation:

Here, we use the out object to display the output on the console. The **System.out.println()** statement executes and prints "Hello" to the console.

Ways to Read Input from the Console in Java:

1. Approach 1: Using System.in.read()

Just like we use out to print output to the console, we can use in (another object of the System class) to take input from the user. The (in) object is associated with the **InputStream** class, we use the **read()** method, which returns an int value and throws a **IOException** (checked exception). Thus, we must handle this exception.

Example:

```java
import java.io.IOException;

public class Hello {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a number:");
        int num = System.in.read();
        System.out.println(num);
    }
}
```

Output:

```
Enter a number:
5  -> User input on console
53 -> Output
```

Explanation:

When we as a user gives input '5', the output is 53 because the **System.in.read()** method returns the **ASCII** (American Standard Code for Information Interchange) value of the input. The ASCII value of '5' is 53.

- **Example:** If the user inputs 'a', the output will be 97, as it is the ASCII value of 'a'.

Example for Character Input:

```java
import java.io.IOException;

public class Hello {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a number:");
        int num = System.in.read();
        System.out.println(num-48);
    }
}
```

Output:

```
Enter a number:
5
5
```

To get the actual number entered, we subtract the ASCII value of '0' (which is 48). For example:

num - 48 = actual number

Thus, if the input is 53 (for '5'), we do:

53 - 48 = 5

This approach reads one character at a time, returning its ASCII value.

---

2. Approach 2: Using BufferedReader

To read multiple characters or entire lines of input, we use the **BufferedReader** class from the *java.io package*. This class provides the readLine() method, which reads a complete line of text and returns it as a string.

Example:

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Hello {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a number:");
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(in);
        int num = Integer.parseInt(bf.readLine());
        System.out.println(num);
    }
}
```

Output:

```
Enter a number:
555
555
```

->user input

->output

Explanation:

Here, we create an **InputStreamReader** object to read from System.in, which is passed to the **BufferedReader** constructor. The readLine() method returns a string, and since we want an integer, we convert the input using Integer.parseInt() which will parse the string into integer.

Closing Resources:

When using BufferedReader (or any resource that reads files or data from external sources), it is good practice to close the resource after use to avoid potential memory leaks. Even though the compiler may not give you errors , failing to close resources can lead to issues such as file locks or security vulnerabilities.

---

3. Approach 3: Using Scanner

In **Java 1.5**, the **Scanner** class was introduced, which became the preferred way to read user input. It is simpler and offers convenient methods to parse primitive types (e.g., nextInt(), nextFloat()) as well as strings.

Example:

```java
import java.util.Scanner;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Enter a number:");
        Scanner sc = new Scanner(System.in);
        int num = sc.nextInt();
        System.out.println(num);
    }
}
```

Output:

```
Enter a number:
456
456
```

->user Input

->output

Explanation:

- We create a Scanner object and use the **nextInt()** method to retrieve the integer input. The Scanner class automatically handles converting the input to the desired type (integer, string, etc.).
- This is currently the most popular approach for reading input in Java.

Comparison of BufferedReader and Scanner:

Thread Safety:

- **BufferedReader:** Synchronized (thread-safe).
- **Scanner:** Not synchronized (not thread-safe).

If you're working in a **multi-threaded** environment, BufferedReader is the better option.

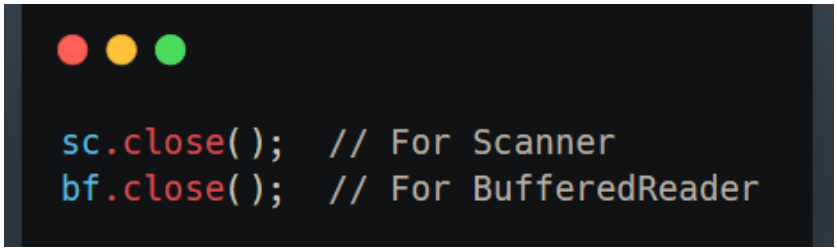
Buffer Size:

- **BufferedReader:** Default buffer size is 8 KB.
- **Scanner:** Default buffer size is 1 KB.

BufferedReader offers better performance for reading large amounts of data or long strings. You can also specify the buffer size when creating a BufferedReader.

Closing Input Streams:

Always remember to close the BufferedReader or Scanner object after use to prevent memory leaks. Example:



11.9-try with resources

Try-with-Resources and Working with finally Block in Java

Working with try and finally Block

In Java, when working with resources (such as file streams, database connections, Networks etc.), it is essential to close the resource after completing its task. **Failing** to close a resource **may result in memory leaks** or prevent other parts of the application from accessing that resource.

The usual approach to ensure that resources are closed is by using a try block with a finally block. The finally block **guarantees** that the statements within it will be executed, whether an error occurs or not. This is especially useful for closing resources like file streams, which must be closed regardless of the outcome of the try block.

Key Points about the finally Block:

- It always executes, even if an exception is thrown.
- It ensures that critical operations, such as closing resources, are performed.
- It works with or without the catch block.

Example 1: Using try, catch, and finally

```java
public class Hello1 {
    public static void main(String[] args) {
        int i = 2;
        int j = 0;
        try {
            j = 18 / i;
        } catch(Exception e) {
            System.out.println("Something went wrong");
        } finally {
            System.out.println("Bye!");
        }
    }
}
```

Output:

```
Bye!
```

Explanation:

In the above example, even though no exception occurs, the finally block is executed.

Example 2: Handling Exceptions with finally

```
public class Hello1 {
    public static void main(String[] args) {
        int i = 0;
        int j = 0;
        try {
            j = 18 / i;
        } catch(Exception e) {
            System.out.println("Something went wrong");
        } finally {
            System.out.println("Bye!");
        }
    }
}
```

Output:

```
Something went wrong
Bye!
```

Explanation:

Here, when the exception occurs, the flow jumps to the catch block, but the finally block still executes. This ensures that necessary actions like closing resources are performed even in the case of an exception.

Example 3: Using finally to Close Resources

```java
public class Hello {
    public static void main(String[] args) throws IOException {
        int num = 0;
        BufferedReader br = null;
        try {
            System.out.println("Enter your number: ");
            InputStreamReader in = new InputStreamReader(System.in);
            br = new BufferedReader(in);
            num = Integer.parseInt(br.readLine());
            System.out.println("You entered: " + num);
        } finally {
                br.close();  // Closing resource
            }
    }
}
```

Explanation:

In this example, the **BufferedReader** resource is closed in the finally block to ensure it is always closed, regardless of whether an exception occurs.

Try-with-Resources Feature in Java

Java provides an alternative approach called **Try-with-Resources** to manage resources efficiently without needing a finally block to close them explicitly. This approach ensures that the resources are automatically closed after the try block completes.

Key Points about Try-with-Resources:

- The try-with-resources statement ensures that each resource is closed automatically at the end of the statement.
- Any object that implements **java.lang.AutoCloseable** (like BufferedReader, FileInputStream, etc.) can be used in this structure.
- It simplifies **resource management** and reduces the risk of resource leaks.

Syntax of Try-with-Resources:

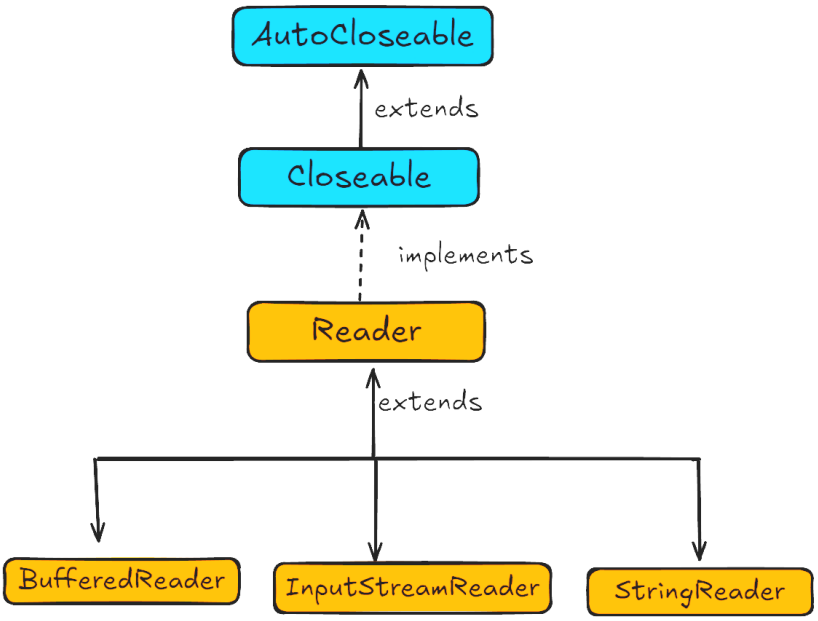
```
try (declare resources here) {
    // use resources
} catch(Exception e) {
    // handle exceptions
}
```

Example: Try-with-Resources

```
public class Hello {
    public static void main(String[] args) throws IOException {
        int num = 0;

        // Using try-with-resources
        try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {
            System.out.println("Enter your number: ");
            num = Integer.parseInt(br.readLine());
            System.out.println("You entered: " + num);
        }
    }
}
```

Explanation:

In the above example, the BufferedReader is declared in the try block, and it automatically closes after the execution of the block. Since BufferedReader implements interface **Closeable**, which extends the another interface **AutoCloseable**, Java ensures it is closed without requiring a finally block.



Advantages of Try-with-Resources:

- **Cleaner code**: No need for explicit resource closing in finally.
- **Automatic resource management**: Resources are automatically closed once the try block completes.
- **Fewer bugs**: Reduces the chances of forgetting to close a resource and causing memory leaks

Conclusion

Using the try with resources statement is the recommended practice for managing resources in Java, as it automatically handles the closing of resources. However, using a finally block provides a clear structure for resource management and can be beneficial in scenarios where specific actions need to be performed regardless of exceptions.

TELUSKO

12.1-What are Threads?

Introduction to Threads and Multithreading

When we run an application on a computer device, such as a laptop or desktop, it operates within the **Operating System (OS)**. The OS, in turn, interacts with the hardware layer, which includes components like:

- **CPU (Central Processing Unit)**: Responsible for executing instructions and performing arithmetic and logical operations.
- **RAM (Random Access Memory)**: Stores data temporarily for quick access.
- **ROM (Read-Only Memory)**: ROM is a type of non-volatile memory, meaning it retains its contents even when the power is turned off. It is an essential component in computer systems and embedded devices for storing critical data and instructions also Often include firmwares which are needed while booting.

The **CPU** is primarily responsible for executing tasks, such as performing arithmetic operations. For example, if we write a program to add two numbers, the CPU executes this operation.

- **Understanding Time Sharing and Multi-Tasking**

Modern operating systems support **multi-tasking**, which means multiple tasks can run simultaneously. When multiple tasks are executed on the OS at the same time, they follow a principle called **time sharing**. Time sharing means that each task receives a specific amount of CPU time to run, process, and execute before switching to another task.

This time-sharing mechanism ensures that all tasks are processed efficiently, making the system more responsive.

What is a Thread?

The smallest and lightest unit of execution within a process is called a **Thread**.

A process constitutes a program that is currently executing. It denotes an active instance of a program executing on a computer. Upon executing a program, the operating system generates a process to oversee and execute it. Each process

functions autonomously and serves as the fundamental unit of resource management in contemporary operating systems.

A **thread** is a single sequence of execution within a program. It is sometimes referred to as a **lightweight process** because it shares the same resources as the main process but operates independently.

Each thread belongs to a single process, and in an operating system that supports **multithreading**, a process can have multiple threads running simultaneously.

Key Points About Threads:

- Threads perform multiple tasks by dividing them into smaller units, allowing for smooth program execution.
- A thread is a single, sequential stream within a process.
- Each thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads are useful in systems with multiple CPUs because they can run independently on different cores. However, on a single-core system, threads need to context switch, which can cause overhead.

Why Do We Need Threads?

Threads are essential for the following reasons:

- **Parallel execution**: Threads allow a process to run multiple tasks simultaneously, improving the performance and responsiveness of an application.
- **Shared Data**: Threads share common data and resources, reducing the need for complex inter-process communication.
- **Reduced Resource Consumption**: Threads are lightweight compared to full processes, as they share the same memory and resources.
- **Efficient Utilization**: Threads make efficient use of system resources, as multiple threads can share the same memory space, making efficient use of system resources.

Example: Multi-Threading in Real Life

In a web browser, each tab can be considered a separate thread. Similarly, in an application like MS Word, multiple threads may be running simultaneously:

- **Thread 1**: Formatting the text.
- **Thread 2**: Processing user inputs.
- **Thread 3**: Checking spelling and grammar in real-time.

Each of these threads runs independently, making the application responsive and efficient.

What is Multi-Threading?

Multi-threading is a technique that allows a process to be divided into multiple threads, enabling parallelism and efficient utilization of CPU resources. This technique helps in achieving **concurrent execution** of multiple parts of a program.

Key Features of Multi-Threading:

- **Parallelism**: Multiple threads can run simultaneously, allowing tasks to be performed faster.
- **Resource Sharing**: All threads within a process share the same memory and resources.
- **Improved Responsiveness**: Multi-threading makes applications more responsive, especially when performing tasks like file I/O, network communication, and user interactions.

Example: Multi-Threading in a Video Game

Consider a video game like a soccer match. In the game:

- One thread handles the **player's movements**.
- Another thread manages the **background audio**.
- A separate thread controls the **graphics rendering**.

TELUSKO

- Additional threads handle the **audience cheering** and **ball physics**.

Each of these tasks operates as a separate thread, allowing the game to run smoothly and respond quickly to user inputs.

By using threads, the operating system achieves a **multi-tasking environment**, where multiple tasks are executed efficiently without noticeable lag.

Conclusion

Threads and multi-threading are essential concepts for improving application performance and responsiveness. They allow processes to perform multiple tasks in parallel by breaking them into smaller, independent units. By understanding and implementing threads effectively, developers can build applications that are fast, efficient, and capable of handling complex operations simultaneously.

12.2-Mutliple Threads

Understanding Threads in Java

In the initial stages of learning about threads, we typically create them manually to understand the core concepts. However, as we delve into more advanced topics, we'll encounter frameworks and libraries that automate thread management, allowing developers to focus solely on building optimized business logic.

Let's start by exploring the basics using a simple example.

Example without Threads

```
 1.  class Hi{
 2.      public void show() {
 3.          for (int i = 0; i < 10; i++) {
 4.              System.out.println("Hi, I'm from class Hi");
 5.          }
 6.      }
 7. }
 8.
 9. class Hello{
10.     public void show() {
11.         for (int i = 0; i < 10; i++) {
12.             System.out.println("Hello, I'm from class Hello");
13.         }
14.     }
15. }
16.
17. public class Demo {
18.     public static void main(String[] args) {
19.         Hi obj1 = new Hi();
20.         Hello obj2 = new Hello();
21.
22.         obj1.show(); // Call method of class Hi
23.         obj2.show(); // Call method of class Hello
24.     }
25. }
```

Output:

```
Hi, I'm from class Hi
Hi, I'm from class Hi
Hi, I'm from class Hi
...
Hi, I'm from class Hi
Hello, I'm from class Hello
Hello, I'm from class Hello
...
Hello, I'm from class Hello
```

Explanation

In the above example:

- Class Each class, Hi and Hello, has a show() method that prints a message ten times.
- In the Demo class, we create objects of both classes and call their show() methods sequentially.
- As a result, the messages from **Class Hi** are printed first, followed by the messages from **Class Hello**.

Observation:
Both classes are executing their tasks sequentially, one after the other. If we want them to run simultaneously and in parallel, we need to use ***threads*.**

Implementing Multiple Threads

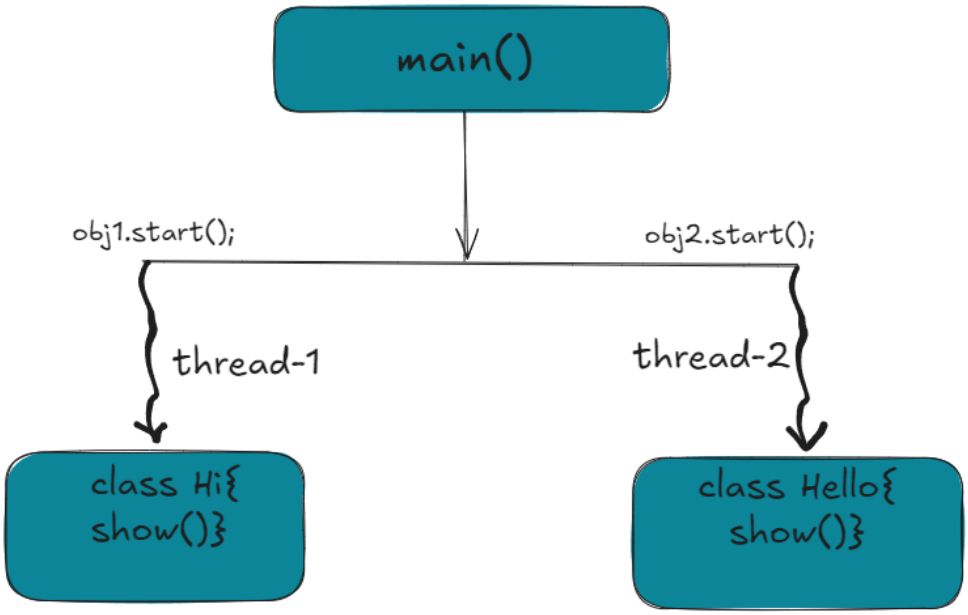
We can modify the above example to run the tasks concurrently using threads. Let’s see how we can achieve this.

Example with Threads

```
class Hi extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hi, I'm from class Hi");
        }
    }
}

class Hello extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello, I'm from class Hello");
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        Hi obj1 = new Hi();
        Hello obj2 = new Hello ();

        obj1.start(); // Start thread for class Hi
        obj2.start(); // Start thread for class Hello
    }
}
```

Explanation:

1. **Extending the Thread Class**:
    - Instead of creating a regular class, we extend the Thread class to make Class A and Class B into threads.
2. **Implementing the run() method**:
    - Each class overrides the run() method to specify the task that each thread should complete.
3. **Starting the Threads**:
    - Instead of calling the show() method directly, we invoke the start() method on both thread objects (obj1.start() and obj2.start()).
    - Internally, the start() method calls the run() method, initiating the thread's execution.



Output with Threads:

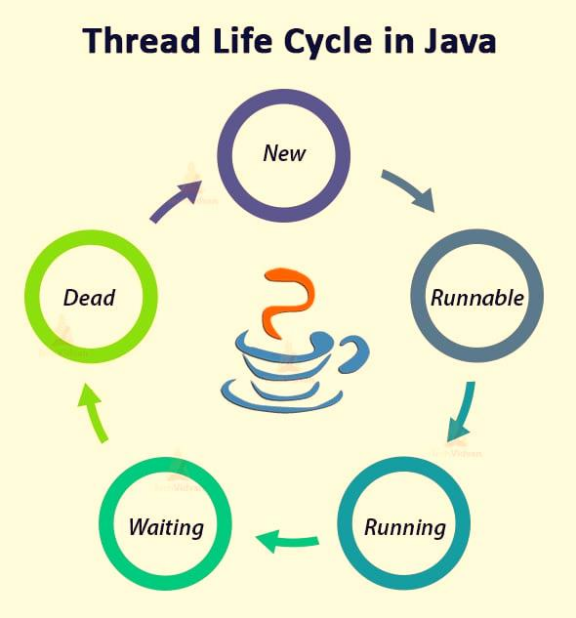
***With threading enabled, you may see an interleaved output like*:**

```
Hi, I'm from class Hi
Hello, I'm from class Hello
Hi, I'm from class Hi
Hello, I'm from class Hello
...
```

This shows that both threads are running concurrently.

Understanding the Role of the Thread Scheduler

The Thread Scheduler, a component operating in the background, determines the execution order of threads. Here's how it works:



- The **Thread Scheduler** is responsible for deciding which thread should run and when.
- If multiple threads are in the ***runnable*** state, the scheduler picks one based on various factors, such as priority and time-sharing principles.
- On a dual-core CPU, two threads can execute simultaneously. If there are

more threads than cores, the scheduler uses a time-sharing mechanism to allocate execution time for each thread.

Real-World Scenario: Multi-Core Systems

Modern computers often have multiple cores (e.g., dual-core, quad-core, or even 16-core CPUs), which allows them to run several threads in parallel. For example:

- Two threads will run simultaneously if a dual-core CPU has six threads to execute.
- Due to time-sharing, the threads will switch between cores, ensuring that each thread gets a chance to execute without remaining idle.

Advantages of Java Multithreading

1. **Improved Performance**:
   Multiple threads allow us to perform various tasks simultaneously, improving overall application performance.
2. **Better Resource Utilization**:
   Multithreading enables better CPU utilization by allowing idle threads to take over tasks when active threads are blocked or waiting.
3. **Non-Blocking Execution**:
   If an exception occurs in one thread, it won't impact the execution of other threads, ensuring a smooth and non-blocking user experience.

Conclusion

Multithreading in Java is a powerful feature that allows developers to create responsive and efficient applications. While creating and managing threads manually is useful for understanding the basics, advanced applications often rely on frameworks and tools that handle thread management automatically. By leveraging multithreading, we can ensure our programs make the most out of modern multi-core systems.

12.3 Thread Priority and Sleep in Java

Understanding the concepts of thread priority and the sleep mechanism is essential for managing how threads execute concurrently in Java. This section covers how to assign priorities to threads and how to control their execution using the sleep() method.

---

1. Introduction to Thread Priority

When Java assigns tasks to multiple threads, they often need to work simultaneously in parallel. However, the thread scheduler determines the order in which threads execute, sometimes leading to unexpected results.

Let’s consider the following example:

```
class A extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hi, I'm from class A");
        }
    }
}

class B extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello, I'm from class B");
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj1 = new A();
        B obj2 = new B();

        obj1.start();
        obj2.start();
    }
}
```

Output:

```
Hi, I'm from class A
Hi, I'm from class A
Hi, I'm from class A
...
Hello, I'm from class B
Hello, I'm from class B
...
```

Explanation:

Here, the output shows that the obj1 thread (class A) executes first, followed by the obj2 thread (class B). The thread scheduler prioritized obj1 to finish its execution before allowing obj2 to run.However, this behavior is not guaranteed and may vary across runs

Thread priorities are only a hint to the thread scheduler. On some systems, priorities may not be strictly followed, especially in multi-threaded environments where multiple factors influence scheduling.

Example: On most modern operating systems (e.g., Windows or Linux), the JVM relies on the OS thread scheduler, which may not strictly adhere to Java thread priorities.

Best Practices you can follow:

Program logic shouldn't rely too much on thread priorities because they aren't cross-platform compatible.

For effective thread management, thread synchronization and other concurrency controls (such as locks or ExecutorService) are typically recommended.

---

2. Using Thread Priority to Control Execution

If you want to control the order of execution between threads, you can assign priorities to each thread. By default, every thread in Java has a priority of 5. The priority range is from **1** (lowest) to **10** (highest).

- **Default Priority**: 5
- **Lowest Priority**: 1

- **Highest Priority**: 10

You can get the current priority of a thread using the getPriority() method:

```
System.out.println(obj1.getPriority() + " " +
obj2.getPriority());
```

This will display the default priority for both obj1 and obj2:

```
5 5
```

Example: Setting Thread Priority

Let's see an example where we manually set the priorities for the threads:

```
public class Demo {
    public static void main(String[] args) {
        A obj1 = new A();
        B obj2 = new B();

        // Set thread priorities
        obj1.setPriority(1);  // Lowest priority
        obj2.setPriority(10); // Highest priority

        obj1.start();
        obj2.start();
    }
}
```

Output:

```
Hello, I'm from class B
Hello, I'm from class B
...
Hi, I'm from class A
Hi, I'm from class A
...
```

Explanation:

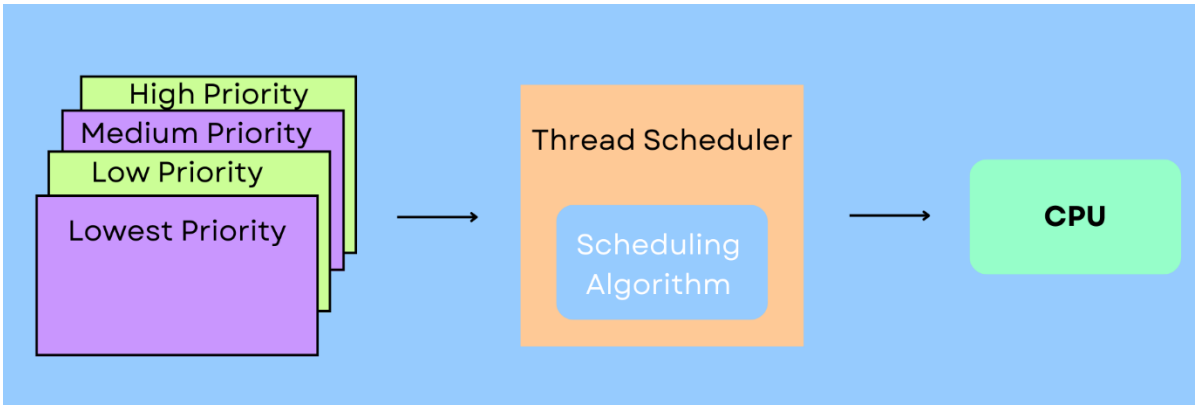
By assigning obj1 a priority of 1 (lowest) and obj2 a priority of 10 (highest), we give obj2 a better chance to execute first. However, the thread scheduler's behavior is platform-dependent, and output may not be guaranteed.

---

3. Thread Priority Constants

Instead of using numeric values for priorities, you can use the predefined constants in the Thread class:

- **Thread.MIN_PRIORITY**: 1
- **Thread.NORM_PRIORITY**: 5
- **Thread.MAX_PRIORITY**: 10



Example: Using Thread Constants

```
public class Demo {
    public static void main(String[] args) {
        A obj1 = new A();
        B obj2 = new B();

        // Using Thread priority constants
        obj1.setPriority(Thread.MIN_PRIORITY);  // 1
        obj2.setPriority(Thread.MAX_PRIORITY);  // 10

        obj1.start();
        obj2.start();
    }
}
```

---

4. Making Threads Sleep

The sleep() method in Java is used to pause the execution of a thread for a specified period. This can help manage thread execution and achieve consistent results.

Syntax:

```
Thread.sleep(milliseconds);
```

- **milliseconds**: The duration in milliseconds for which the current thread will sleep.

Important Points:

1. **Throws InterruptedException**: The sleep() method throws an InterruptedException, which is a checked exception that must be handled using a try-catch block.
2. **Pauses Current Thread**: As Thread.sleep() is a static method, it affects only the current executing thread by pausing the current thread without releasing any locks it holds, meaning other threads waiting for the same lock cannot proceed until the sleeping thread completes or exits the synchronized block.

Example: Using sleep() Method

```
class A extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hi, I'm from class A");

            // Make the thread sleep for 1 second
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " +
e);
            }
        }
    }
}

class B extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hello, I'm from class B");
```

```
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj1 = new A();
        B obj2 = new B();

        obj2.start();
        obj1.start();
    }
}
```

Output:

```
Hi, I'm from class A
Hello, I'm from class B
Hello, I'm from class B
...
Hi, I'm from class A
Hi, I'm from class A
...
```

Explanation:

Here, obj1 prints "Hi,I'm **from** class A " and then sleeps for 1 second before printing again. Meanwhile, obj2 can execute without any delay, leading to interleaved outputs.

12.4-Runnable vs Thread

Introduction to the Runnable Interface

The Runnable interface in Java is a functional interface that contains a single abstract method, **run()**. Because it only has one method, you often use it in scenarios where you want to execute code concurrently in a separate thread without inheriting from the Thread class.

Why Use Runnable?

When we want to create a new thread, we usually call the start() method. Internally, the **start()** method of the Thread class which invokes the run() method, and it does so by expecting a Runnable instance. This allows us to decouple the task logic from the actual thread implementation.

Example Scenario: Multiple Inheritance Issue

Let's consider a scenario where you want to create a class Z as the parent class for class A, but you also want A to perform some concurrent operations using the Thread class. However, Java does **not** support multiple inheritance, so you cannot extend both Thread and Z at the same time.

Incorrect Example:

```
class Z { }
class A extends Z, Thread {  // Not allowed in Java as it does not
support multiple inheritance
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hi, I'm from class A");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e);
            }
        }
    }
}
```

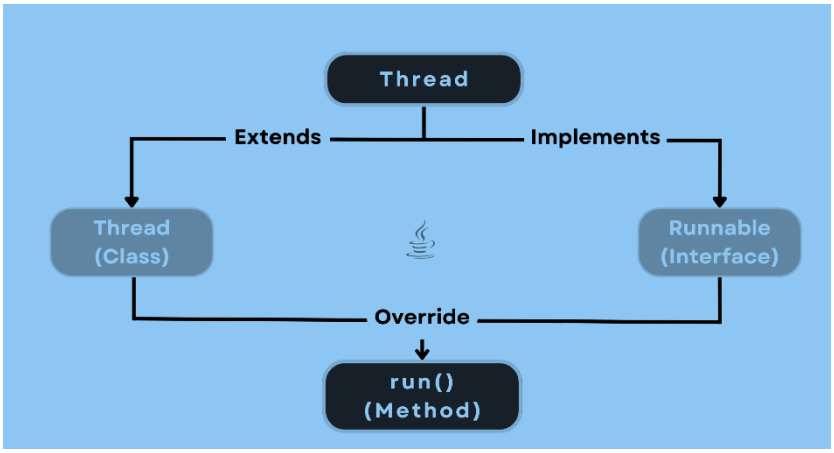
Solution: Using the Runnable Interface

Instead of extending Thread, we implement the Runnable interface. This way, we can still extend the class Z while also implementing Runnable to achieve the desired functionality.

```
class Z { }

class A extends Z implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hi, I'm from class A");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " + e);
            }
        }
    }
}
```

Runnable vs. Thread: A Better Approach

The above implementation allows class A to extend its parent class Z and simultaneously perform tasks in a new thread by implementing Runnable also It is important to note that calling start() initiates a new thread, which in turn calls the run() method. Directly invoking run() will execute the task sequentially in the main thread, contradicting the purpose of multithreading.
Let's extend the example to demonstrate the use of two classes implementing Runnable:



Example: Two Runnable Implementations

```
class A implements Runnable {
        public void run() {
    for (int i = 0; i < 5; i++) {
System.out.println("Hi, I'm from class A");
                    try {
                Thread.sleep(1000);
    } catch (InterruptedException e) {
System.out.println("Thread interrupted: " + e);
                    }
                }
            }
        }

class B implements Runnable {
        public void run() {
    for (int i = 0; i < 5; i++) {
System.out.println("Hello, I'm from class B");
                    try {
                Thread.sleep(1000);
    } catch (InterruptedException e) {
System.out.println("Thread interrupted: " + e);
                    }
                }
            }
        }

        public class Demo {
    public static void main(String[] args) {
            Runnable obj1 = new A();
            Runnable obj2 = new B();

Thread t1 = new Thread(obj1);  // Creating thread t1
Thread t2 = new Thread(obj2);  // Creating thread t2

        t1.start();  // Start thread t1
        t2.start();  // Start thread t2
                }
            }
```

Output:

```
Hi, I'm from class A
Hello, I'm from class B
Hi, I'm from class A
Hello, I'm from class B
...
```

Explanation:

1. The class A and B implement the Runnable interface.
2. We create two separate threads (t1 and t2) using instances of Runnable implementations (obj1 and obj2).
3. This approach allows us to have more flexible designs by decoupling the thread execution from class inheritance.

Optimizing Code with Anonymous Classes and Lambda Expressions

Sometimes, creating separate classes just to implement the Runnable interface can make the code lengthy and less readable. To solve this, we can use **Anonymous Classes** and **Lambda Expressions** to optimize our code.

Using Anonymous Classes

```
public class Demo {
    public static void main(String[] args) {
        Runnable obj1 = new Runnable() {
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println("Hi, I'm from class A");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        System.out.println("Thread interrupted: " +
e);
                    }
                }
            }
        };

        Runnable obj2 = new Runnable() {
            public void run() {
                for (int i = 0; i < 5; i++) {
                    System.out.println("Hello, I'm from class B");
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        System.out.println("Thread interrupted: " +
```

```
e);
                    }
                }
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

        t1.start();
        t2.start();
    }
}
```

Using Lambda Expressions

Since Runnable is a functional interface, we can simplify the code using **lambda expressions**:

```
public class Demo {
    public static void main(String[] args) {
        Runnable obj1 = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Hi, I'm from class A");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("Thread interrupted: " + e);
                }
            }
        };

        Runnable obj2 = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Hello, I'm from class B");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println("Thread interrupted: " + e);
                }
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);
```

```
        t1.start();
        t2.start();
    }
}
```

Explanation of Lambda Optimization:

- **Lambda expressions** reduce code verbosity and make the implementation more concise.
- This approach is highly recommended when implementing single-method interfaces like Runnable.

Differences Between Thread and Runnable

1. Extending vs. Implementing:

- **Thread**: If a class extends Thread, it cannot extend any other class due to Java's single inheritance constraint.
- **Runnable**: A class can implement multiple interfaces, allowing more flexibility.

2. Reusability:

- **Thread**: The task and the thread are tightly coupled, which can limit reusability.
- **Runnable**: Separates the task from the thread, promoting better object-oriented design and modularity.

3. Resource Sharing:

- **Runnable**: Promotes resource sharing as a single Runnable instance can be shared among multiple threads.
- **Thread**: Each thread has its own instance, which can lead to increased memory consumption.

Conclusion

In summary, both the Thread class and the Runnable interface play crucial roles in Java's multithreading model. Developers should carefully evaluate their

application requirements to decide between the two. While Thread offers straightforward usage, Runnable is typically preferred due to its flexibility, reusability, and better support for resource sharing.

12.5: Race Condition

Introduction:

Threads play an essential role in multitasking and concurrent execution in programming. As we've discussed, you can either manually create threads or rely on frameworks to provide them. Although creating threads manually is not always common, sometimes it's necessary, especially when you want a specific task to execute concurrently to improve performance.

We can create threads to handle tasks that require rapid completion, depending on the requirements. In such cases, we might manually create two or more threads at the same time. However, this introduces some challenges, such as synchronization issues and race conditions.

Understanding Mutation:

Mutation, in simple terms, means changing the state or value of something. In programming, mutation occurs when a variable's value is altered during the execution.

For example, consider a variable i that has a value of 10. If we change it to 12, it is referred to as a mutation of that variable. In the context of threads, a race condition arises when two or more threads access and change the same variable simultaneously.

This results in **inconsistent values** due to unpredictable behavior, making the scenario unstable.

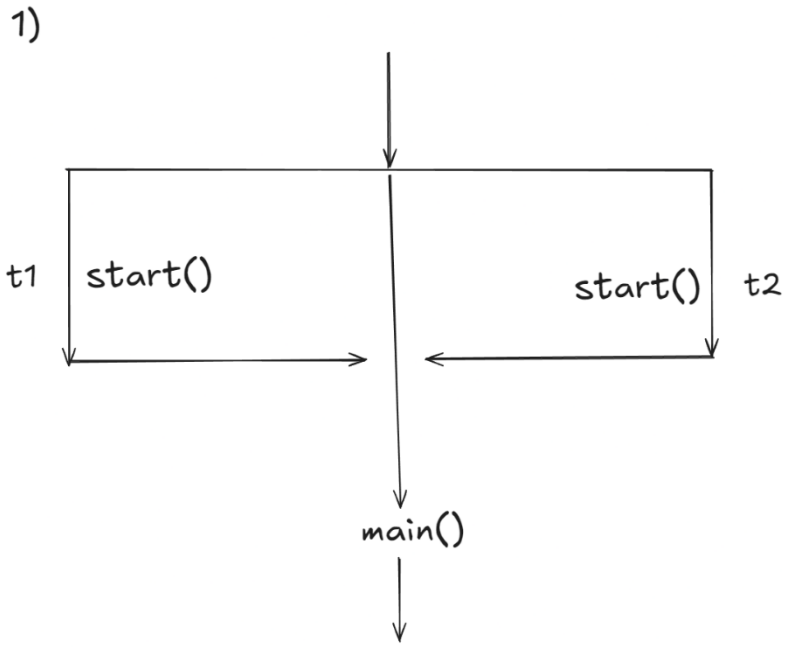
Let's illustrate this with a simple analogy:

Example:

Suppose you are going to an ATM to withdraw money. Imagine there is only one entrance to the ATM booth, and only one person can enter at a time. Now, if you and another person arrive at the same time and try to enter, what would happen? You won't both be able to get in simultaneously. This situation mirrors the behavior of threads when they attempt to access the same resource simultaneously, causing a conflict.

This is why, when working with threads, we should avoid using **mutable variables**. Alternatively, we can use **immutable data** or ensure that the methods manipulating shared data are **thread-safe**.

Understanding the join() Method:



The **join()** method ensures that the main thread waits for the other threads (t1 and t2) to finish execution before proceeding. This is necessary to ensure that the final value of count is printed after both threads have completed their tasks.

Fig(1).without implementation of join() method

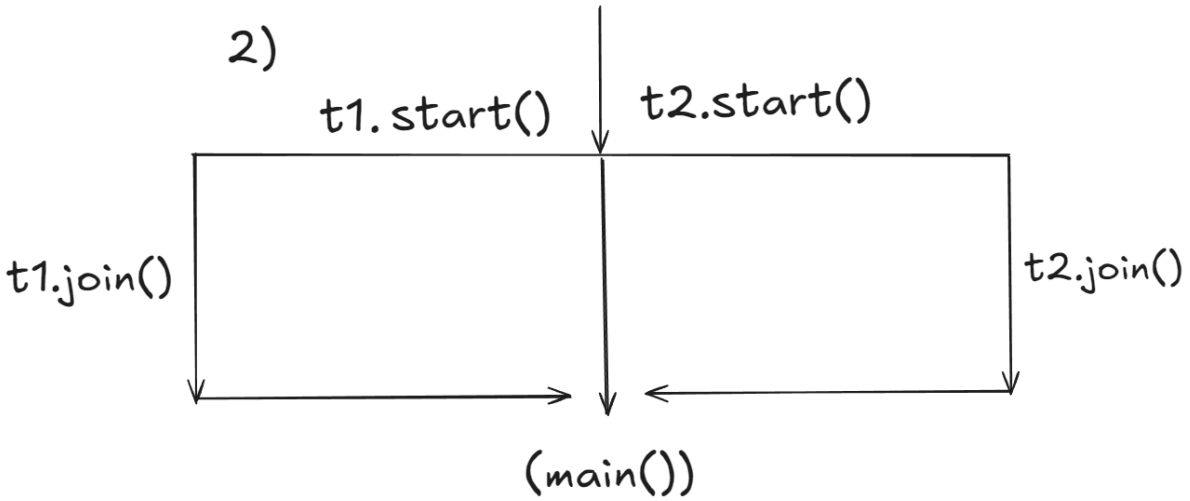


Fig. 2 with implementation of join() method

What is Thread Safety?

Thread safety means ensuring that a method or a block of code is executed by only one thread at a time. This prevents conflicts and inconsistencies. When a

method is thread-safe, it ensures that no other thread can access or modify the shared resource while it's in use by one thread.

Let's explore this further using an example.

Example without Thread Safety:

```
class Counter {
    int count;

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

public class Demo {
    public static void main(String[] args) throws
InterruptedException {
        Counter c = new Counter();

        Runnable obj1 = () -> {
            for(int i = 1; i <= 1000; i++) {
                c.increment();
            }
        };

        Runnable obj2 = () -> {
            for(int i = 1; i <= 1000; i++) {
                c.increment();
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println(c.count);
    }
}
```

Output (varies each time the program is executed):

- 1750
- 1985
- 2000
- 1905

Explanation:

- In the above program, we created a Counter class with an increment() method that increases the value of the variable count.
- We then created two threads (t1 and t2) that execute the increment() method 1000 times each.
- Ideally, the output should be 2000, but due to race conditions, the threads interfere with each other, resulting in an inconsistent count value.

The inconsistency occurs because both threads access the shared variable count simultaneously. To solve this issue, we need to make the increment() method **thread-safe** using the synchronized keyword.

Example with Thread Safety:

```
class Counter {
    int count;

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

public class Demo {
    public static void main(String[] args) throws
InterruptedException {
        Counter c = new Counter();

        Runnable obj1 = () -> {
            for(int i = 1; i <= 1000; i++) {
                c.increment();
            }
        };

        Runnable obj2 = () -> {
            for(int i = 1; i <= 1000; i++) {
```

```
                c.increment();
            }
        };

        Thread t1 = new Thread(obj1);
        Thread t2 = new Thread(obj2);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println(c.count);
    }
}
```

Output:

- 2000 (Consistent output)

Explanation:

- By using the synchronized keyword on the increment() method, we ensure that only one thread can access this method at a time.
- This prevents the race condition and results in a consistent value of 2000.

Why Do Race Conditions Happen?

Race conditions happen because of the unpredictability of thread execution. If two threads access the same variable simultaneously, the outcome depends on the order in which they execute. For a small number of iterations, the problem might not be noticeable, but as the number of iterations increases (e.g., 10,000), the issue becomes prominent and can lead to **data loss** or **incorrect values**.

Best Practices to Avoid Race Conditions:

1. **Use the synchronized keyword:** Ensures that only one thread can access the critical section of code.
2. **Avoid shared mutable state:** Wherever possible, use immutable objects.
3. **Use Thread-safe Collections:** Use ConcurrentHashMap or other thread-safe collections instead of regular ones like HashMap.

12.6: Thread States in Java

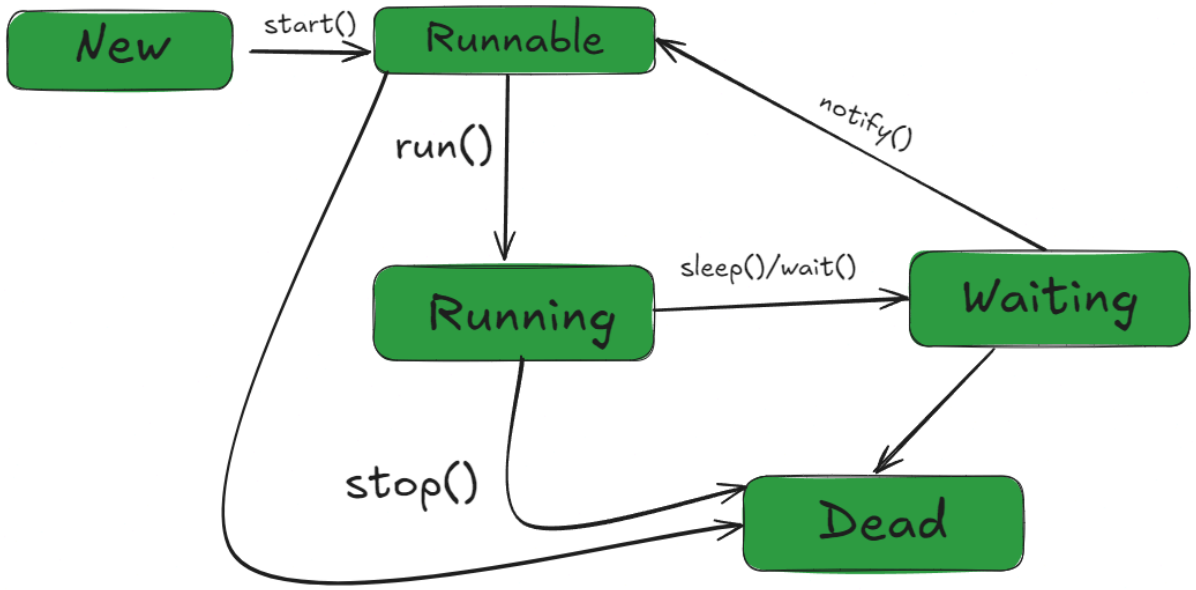
When working with threads in Java, understanding the various states a thread can go through is crucial. A thread, at any given point, can only exist in one of the predefined states. Let's look at the different stages of a thread's lifecycle and how it transitions between them.

Thread Lifecycle States

In Java, a thread exists in one of the following states during its lifecycle:

1. **New State**
2. **Runnable State**
3. **Running State**
4. **Waiting/Blocked State**
5. **Terminated/Dead State**

Each of these states plays an important role in the multithreading environment, allowing Java to manage thread execution and synchronization effectively.



---

1. New State

- **Definition**: When a thread is first created using the Thread class, it enters the **New** state.

- **Description**: In this state, the thread has been initialized but has **not yet started**. The thread is ready for execution but waits for a signal (invocation of the start() method) to begin its lifecycle.
- **Transition**: Once the start() method is called on the thread object, it moves from the **New** state to the **Runnable** state.

Example Code:

```
Thread thread = new Thread(); // Thread is in New State
```

---

2. Runnable State

- **Definition**: After a thread is started, it enters the **Runnable** state.
- **Description**: In this state, a thread is **ready to run** and waiting for CPU allocation. It could be actively running or just **waiting for CPU time** to be scheduled.
- **Thread Scheduler Role**: The thread scheduler decides which thread in the runnable pool will get CPU time.
- **Transition**: The thread remains in the **Runnable** state until it is allocated CPU time, at which point it moves to the **Running** state.

Example Code:

```
thread.start(); // Moves the thread to Runnable State
```

---

3. Running State

- **Definition**: The **Running** state is where the thread is **actively executing** its code.
- **Description**: In this state, the CPU has assigned a time slot for the thread, and it's currently executing its task in the run() method.

- **Thread Scheduler**: The thread scheduler periodically allocates CPU time to threads. After a thread's time slice is over, it may go back to the **Runnable** state.

Transition:

- Runnable → Running: The thread starts running when the CPU is allocated to it.
- Running → Runnable: If the thread is **paused** (due to context switching), it returns to the **Runnable** state.

---

4. Waiting/Blocked State

- **Definition**: A thread enters the **Waiting** or **Blocked** state when it is **temporarily inactive** and waiting for some condition to be met.
- **Description**: This happens when a thread calls methods like wait(), sleep(), or is waiting for a resource (e.g., input/output operation) to become available.
  - **Blocked State**: The thread is waiting for a **resource**.
  - **Waiting State**: The thread is **waiting for another thread** to perform a specific action, such as a notification.
- **Transition Back to Runnable**: Once the required condition is met or the waiting period ends (e.g., using notify(), notifyAll(), or the specified sleep time ends), the thread returns to the **Runnable** state.

Example Methods:

- sleep(): Pauses the thread for a specified duration.
- wait(): Waits indefinitely until another thread calls notify() or notifyAll().

Example Code:

```
try {
    Thread.sleep(1000); // Pauses the thread for 1 second
} catch (InterruptedException e) {
    e.printStackTrace();}
```

5. Terminated/Dead State

- **Definition**: A thread enters the **Terminated** or **Dead** state when it has **completed its execution**.
- **Description**: This happens when the run() method completes execution or when the thread is terminated using certain deprecated methods (e.g., stop()).
- **Deprecated Method**: Calling the stop() method can also cause a thread to terminate, but this is **not recommended** as it can cause the program to behave unexpectedly.

Example Code:

```
public void run() {
    System.out.println("Thread is running...");
    // After this method completes, the thread moves to the
Dead State
}
```

Conclusion

Understanding the different states of a thread and their transitions is crucial for effective multithreading in Java. Each state—**New**, **Runnable**, **Running**, **Waiting/Blocked**, and **Terminated**—serves a specific purpose in managing thread execution. A thread moves through these states based on method calls like start(), sleep(), and wait(), as well as through conditions such as CPU allocation and resource availability.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting for satisfying some conditions. Therefore, a thread is always in any of the five states.

By mastering these thread states and transitions, developers can better control thread behavior, optimize program performance, and avoid potential issues such as deadlock or thread starvation. A clear grasp of the thread lifecycle helps in writing robust and efficient multithreaded programs.

13.1 - Collection API

👉🏻 Introduction to Collection API

In Java, the term **Collection API** can be confusing because it involves three related terms: **Collection**, **Collections**, and **Collection API**. Although they sound similar, each has a distinct meaning:

- **Collection API**: Refers to a set of classes and interfaces used to implement data structures like ArrayList, LinkedList, Queue, and more. It provides a framework for handling groups of objects.
- **Collection**: This is an interface that serves as the root of the collection hierarchy. It represents a group of objects, known as elements, and is the parent interface for various data structure classes.
- **Collections**: A utility class containing static methods that operate on collections. It provides various algorithms such as sorting, searching, and shuffling.

The **Collection API** was introduced in **Java 1.2**, and it significantly simplifies the process of working with data. Let's explore why this API is important and what issues it addresses.

👉🏻Why Use the Collection API?

Before the introduction of the Collection API, Java developers mainly relied on arrays to store data. While arrays are useful, they come with some limitations:

- **Fixed Size**: Once an array is declared, its size is fixed and cannot be changed. If more elements need to be added, a new array of a larger size must be created, and existing elements must be manually copied to the new array. This approach is not efficient in terms of time and memory.
- **Homogeneous Data**: Arrays can only store elements of a single data type. For example, if we need to store details about a student (such as name, age, height, and weight), using a single array would be insufficient. While an array of objects can be created, managing multiple objects becomes cumbersome.

- **Complex Operations**: Operations like sorting, inserting, or removing elements in an array require manual implementation, which can be error-prone and time-consuming.

These limitations made arrays less optimal for working with complex data structures. To address these issues, the **Collection API** provides various data structures that are flexible and easier to use.

👉Advantages of the Collection API

The Collection API offers several benefits that make it preferable over arrays:

1. **Dynamic Sizing**: Data structures like ArrayList can grow or shrink in size depending on the number of elements. This dynamic nature helps manage data more efficiently.
2. **Support for Heterogeneous Data**: Unlike arrays, collection classes can store different types of data in a single structure. For instance, an ArrayList can hold a mixture of strings, integers, or other objects.
3. **Built-in Methods for Data Manipulation**: Collection implementing classes comes with built-in methods for operations like sorting, inserting, searching, and deleting, making these tasks simpler.
4. **Efficient Memory Usage**: Collections classes can automatically adjust their capacity, ensuring that memory is used optimally.

👉Key Components of the Collection API

The Collection API consists of multiple classes and interfaces, each serving specific purposes. Here's a brief overview of some commonly used components:

- **List Interface**: Represents an ordered collection (sequence) of elements. Examples include ArrayList, LinkedList, and Vector.
- **Set Interface**: Represents a collection that does not allow duplicate elements. Examples include HashSet, LinkedHashSet, and TreeSet.
- **Map Interface**: Represents a collection of key-value pairs. Examples include HashMap, LinkedHashMap, and TreeMap.
- **Queue Interface**: Represents a collection designed for holding elements prior to processing. Examples include PriorityQueue and ArrayDeque.

Each of these interfaces and their implementations provide different functionalities, allowing developers to choose the appropriate data structure based on their requirements.

👉Comparison with Arrays

The Collection API offers significant advantages over traditional arrays:

Feature	**Arrays**	**Collection API**
Size	Fixed	Dynamic (can grow/shrink)
Data Type	Homogeneous	Can be homogeneous or heterogeneous
Built-in Methods	Limited	Rich set of methods for data manipulation
Performance (Insertion, Deletion)	Manual implementation required	Methods for efficient operations
Memory Management	Requires manual resizing	Automatic resizing

TELUSKO

13.2-ArrayList

👉Introduction to Collections

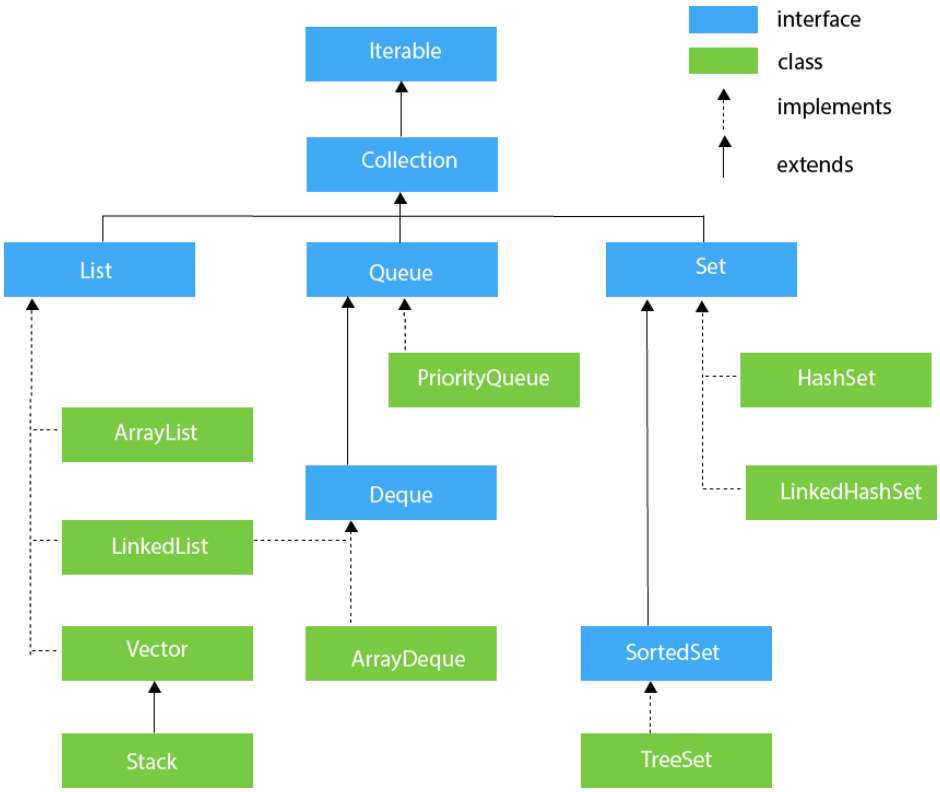
In Java, starting from version 1.2, the Collections Framework was introduced to enhance array operations and provide a set of standard interfaces and classes for handling data. The Collections API includes a variety of interfaces and classes with many built-in methods for data manipulation. To use the Collections Framework, you need to import the ***java.util package***, which contains these interfaces and classes.

Previously, most code did not require importing external packages because the core classes were included in the ***java.lang package***. This package automatically imports classes like Object, the superclass of all classes in Java.

Similarly, the Collections API is part of the ***java.util package***, which is also known as the **utility package** because it provides methods for various tasks such as **sorting** and data **insertion**.

👉Hierarchy of the Collection Interface

The top-level interface in the Collection Framework is **Iterable**, followed by the Collection interface. The Collection interface is then extended by the most



TELUSKO

commonly used interfaces, such as **List, Queue**, and **Set**. Each of these interfaces has its own set of implementing classes, which provide specific functionalities.

- **List Interface:** Implemented by classes such as ArrayList, LinkedList, and Vector.
- **Queue Interface:** Implemented by classes such as PriorityQueue. The **Deque interface**, which extends Queue, is implemented by classes like ArrayDeque.
- **Set Interface:** Commonly used in more advanced topics. It is implemented by classes such as HashSet and LinkedHashSet.
- **Map Interface:** Although not directly under the Collection hierarchy, the Map interface is part of the Java Collections Framework. It stores data in key-value pairs and is implemented by classes like HashMap and LinkedHashMap.

👉Example

Let's look at a simple example using the Collection & List interface with the ArrayList class.

```
import java.util.Collection;
import java.util.ArrayList;

public class Demo {
    public static void main(String[] args) {
        // Using Collection interface with ArrayList implementation
        Collection<Integer> nums = new ArrayList<Integer>();
        nums.add(3);
        nums.add(4);
        nums.add(5);
        nums.add(9);
        nums.add(8);
        nums.add(7);

        // Printing the ArrayList
        System.out.println(nums);
    }
}
```

👉**Output:**

```
[3, 4, 5, 9, 8, 7]
```

👉**Explanation**

In the above example, we use the Collection interface and the ArrayList class. When writing code in an IDE, you might see a warning suggesting using generics for type safety. This is done by specifying the type of data inside angular brackets (< >), which ensures that only a particular data type can be added to the collection. Generics help to prevent errors by avoiding the insertion of incompatible data types.

Since Java treats everything as an object, collections store elements as objects. For primitive types, you should use their corresponding wrapper classes (e.g., Integer for int).

The **add()** method is used to insert elements into the ArrayList. There are several other methods available for collection, which should be explore based on our need as each method as different working.

After adding elements, the ArrayList can be printed directly without using a loop, as the **toString()** method of the collection handles this internally. However, if you want to print each element separately, you can use an enhanced for loop.

Printing Separate Values Using a Loop

```
for (Integer num : nums) {
    System.out.println(num);
}
```

👉**Accessing Elements by Index**

If you need to access elements by their index, use the List interface as the Collection interface doesn't have methods to work with indexing which List provides, an get(int index) method to retrieve the element at a specific position.

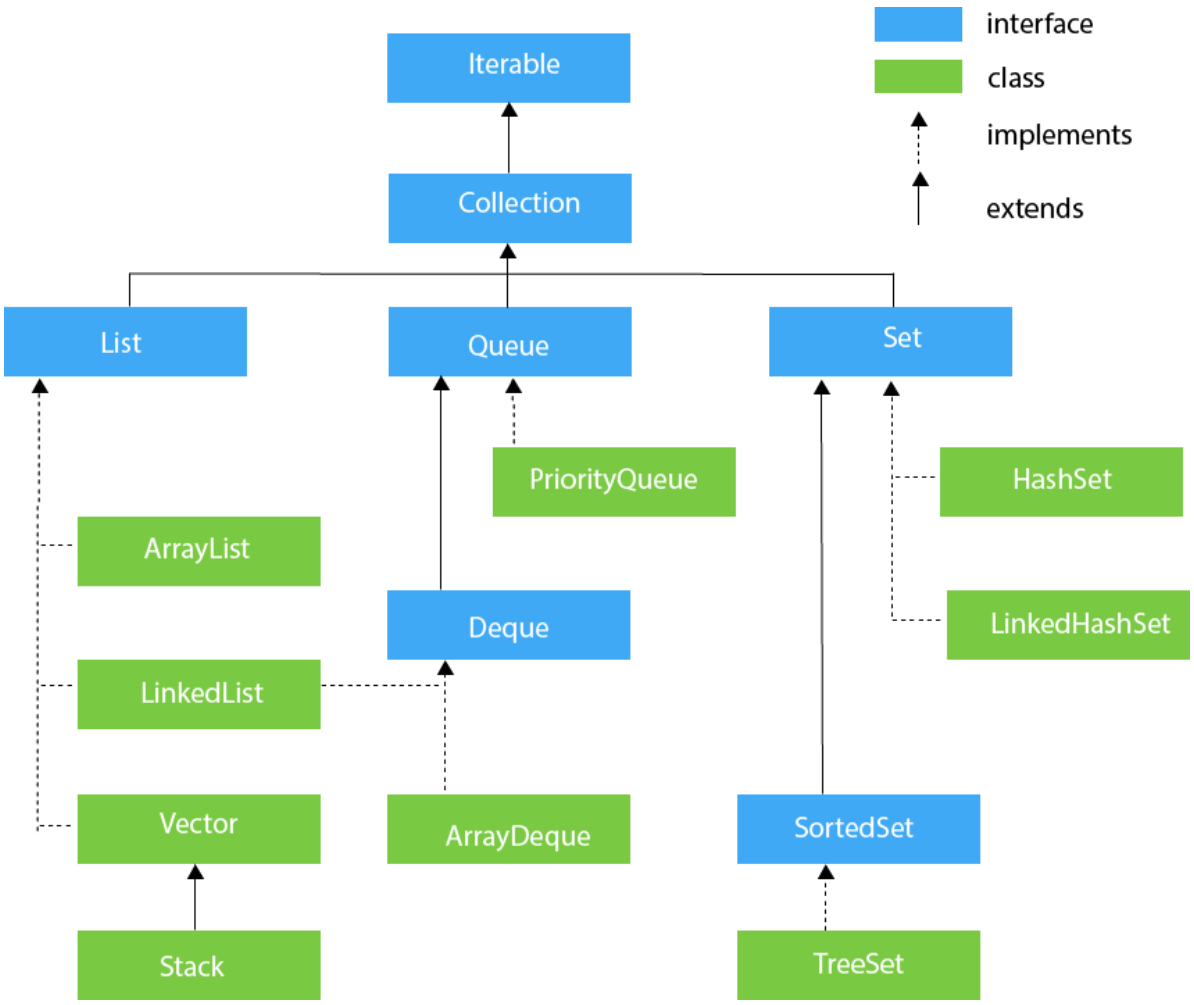
```
List<Integer> numList = new ArrayList<>(nums);
System.out.println("Element at index 2: " + numList.get(2));
```

TELUSKO

13.3 - Set Interface in Java

👉Introduction

In Java, one of the most commonly used interfaces in the Collection framework is the List interface, which allows duplicate elements and supports operations like adding, sorting, and inserting and fetching elements based on index. Two popular classes implementing the List interface are ArrayList and LinkedList. These classes allow you to add duplicate elements and maintain the order in which elements were inserted.

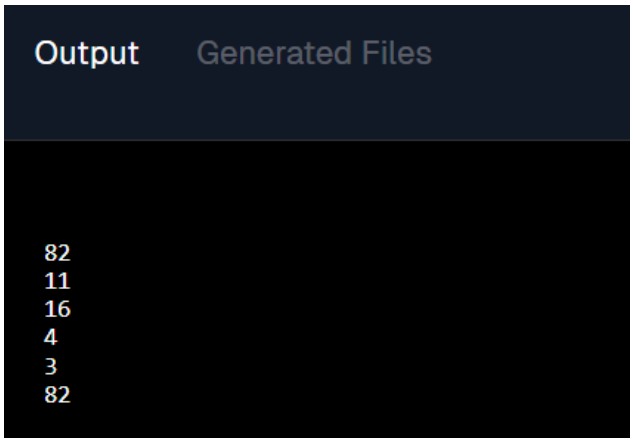


TELUSKO

👉Example of Using a List

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = new ArrayList<Integer>();
        nums.add(82);
        nums.add(11);
        nums.add(16);
        nums.add(4);
        nums.add(3);
        nums.add(82); // Duplicate element

        for (Object num : nums) {
            System.out.println(num);
        }
    }
}
```

👉Output:



In the above example, duplicate elements (82) are allowed. However, if you need to store unique values where duplicates are not permitted, the Set interface is the appropriate choice.

👉Set Interface

The Set interface is a part of the Collection framework in Java, designed to hold a collection of unique elements but it also doesn't supports index based operations as List does . It does not allow duplicate values and provides functionalities to ensure all elements in the collection are distinct.

👉Characteristics of Set Interface:

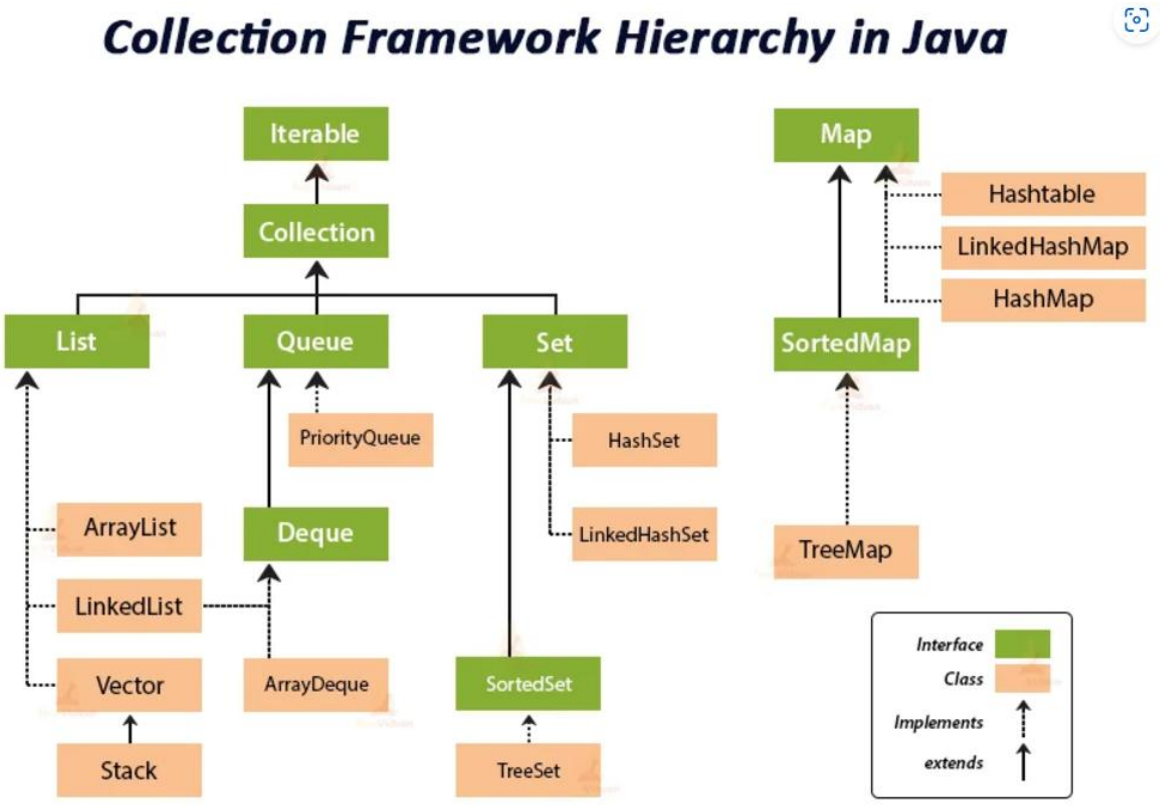
- Extends the Collection interface.
- Guarantees that no duplicate elements are present.
- Implemented by classes like HashSet, LinkedHashSet(For index based operations), and TreeSet(For ordered Sorting of elements).

👉Syntax for Declaring a Set

```
Set<Integer> values = new HashSet<Integer>();
```

👉Implementing Classes of Set Interface

1. **HashSet**:
    - Does not maintain any order of elements.
    - Uses a hashing algorithm for storing elements, making retrieval fast.
2. **LinkedHashSet**:
    - Maintains the insertion order of elements.
    - Slower than HashSet due to the maintenance of the order.
3. **TreeSet**:
    - Stores elements in sorted order (ascending).
    - Implements the NavigableSet interface, which extends the SortedSet interface.
    - Ensures that elements are both unique and sorted.



👉Methods in Set Interface

1. add()

The add() method is used to insert elements into the set. If an element already exists in the set, it will not be added again (no duplicates are allowed).

👉Example: Using HashSet

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        Set<Integer> values = new HashSet<Integer>();
        values.add(82);
        values.add(11);
        values.add(16);
        values.add(4);
        values.add(3);
        values.add(82); // Duplicate element

        for (int num : values) {
            System.out.println(num);
```

```
        }
    }
}
```

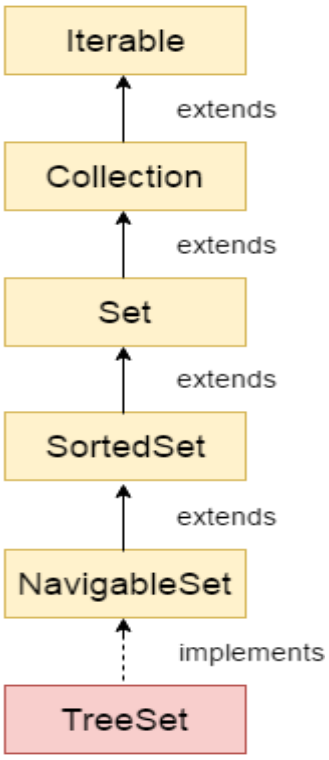
```
Output    Generated Files

16
82
3
4
11
```

👉Explanation:

- The output may not appear in the order in which the elements were added. This is because HashSet does not maintain insertion order.
- Duplicate values (82) are removed automatically as Set Supports only Unique values.

👉Using TreeSet for Sorted and Unique Values

If you need a collection where the elements are both unique and sorted, you should use the TreeSet class. The TreeSet class implements the NavigableSet interface, which extends the SortedSet interface. As a result, elements are stored in a sorted manner.



👉Example: Using TreeSet

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        Set<Integer> values = new
TreeSet<Integer>();
```

```
        values.add(82);
        values.add(11);
        values.add(16);
        values.add(4);
        values.add(3);
        values.add(82); // Duplicate element

        for (int num : values) {
            System.out.println(num);
        }
    }
}
```

👉Output:

```
3
4
11
16
82
```

- The output is sorted in ascending order because of the TreeSet implementation.
- Duplicate values are not added.

👉Understanding the Iterable Interface

The Iterable interface is the top-most interface in the Collection framework, which allows for iterating over a collection. It provides the ability to traverse the elements using an iterator.

👉Example: Using Iterator with TreeSet

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        Set<Integer> values = new TreeSet<Integer>();
        values.add(82);
        values.add(11);
```

```
        values.add(16);
        values.add(4);
        values.add(3);
        values.add(82); // Duplicate element

        Iterator<Integer> iterator = values.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

👉**Output**

```
3
4
11
16
82
```

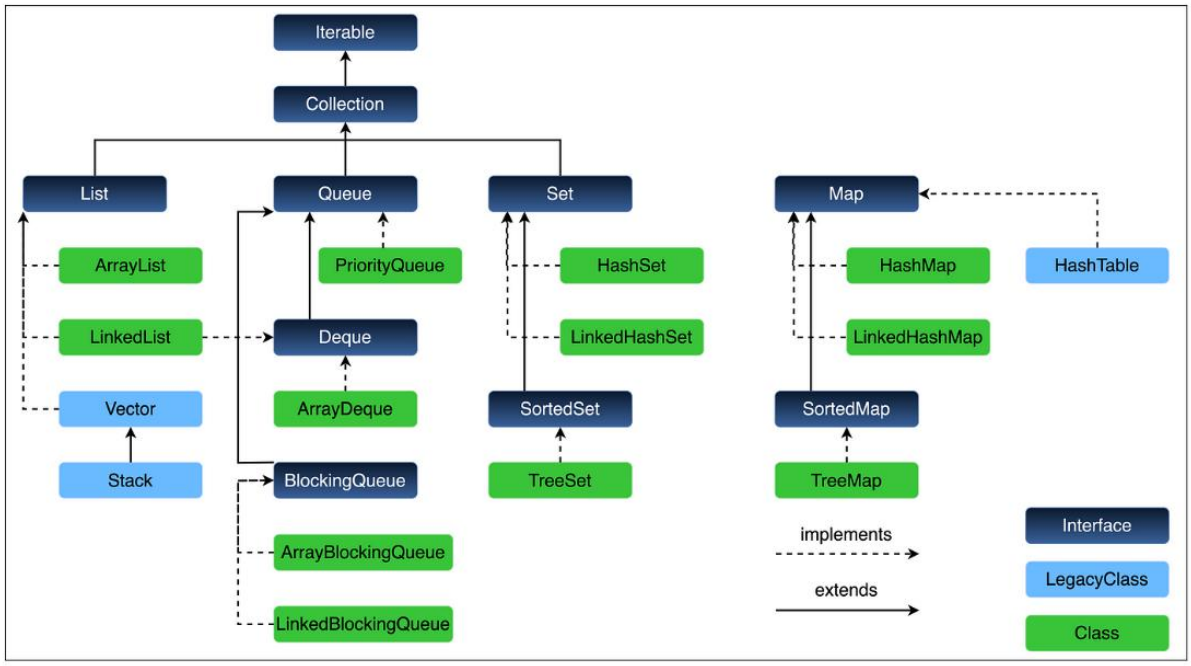
👉**Methods in Iterator**

1. **hasNext()**: Checks if there are more elements in the collection.
2. **next()**: Retrieves the next element in the collection.

TELUSKO

13.4-Map

The Map interface is part of the Java Collection Framework, although it does not extend the Collection interface. In Java, a Map represents a collection of key-value pairs, where each unique key maps to a specific value. This concept is similar to a telephone directory, where a person's name (key) is associated with a phone number (value). When you provide a key, the corresponding value can be retrieved.



👉Key Features of Map Interface

- **Mapping between Keys and Values**: The Map interface allows for the association of unique keys with specific values.
- **Unique Keys**: Each key in a Map must be unique, while values can be duplicated.
- **Not a Subtype of Collection Interface**: Unlike other collection types, Map is not a subtype of the Collection interface, so it behaves differently.

TELUSKO

👉Creating Map Objects

Since Map is an interface, you cannot create objects of the Map type directly. You must use a class that implements the Map interface to create an object, such as HashMap, LinkedHashMap, or TreeMap.

👉Example Syntax for Creating a Map

```
Map<ObjectType1, ObjectType2> mapName = new HashMap<>();
```

- The ObjectType1 represents the type of the keys, while ObjectType2 represents the type of the values.

👉Example: Storing Student Names and Marks

Here's how you can create a Map to store student names (as keys) and their marks (as values):

```
import java.util.Map;
import java.util.HashMap;

public class Demo {
    public static void main(String[] args) {
        Map<String, Integer> students = new HashMap<>();
        students.put("Navin", 56);
        students.put("Harsh", 65);
        students.put("Sushil", 73);
        students.put("Kiran", 96);

        System.out.println(students);
    }
}
```

👉Output:

```
{Navin=56, Harsh=65, Sushil=73, Kiran=96}
```

👉Explanation

- We created a Map object named students using HashMap, specifying the key type as String and the value type as Integer.
- The put() method is used to add key-value pairs to the map.
- When we print the students map, it displays the names and their corresponding marks.

👉Working with Map Elements

- **Accessing Values**: You can retrieve a value from the map using the get() method by providing the corresponding key. For example:

```
System.out.println(students.get("Harsh"));
```

👉**Output**: 65

- **Handling Duplicate Keys**: If you add a key that already exists in the map, the new value will replace the old value. For instance:

```
students.put("Harsh", 45);

System.out.println(students);
```

👉**Output**:

```
{Navin=56, Harsh=45, Sushil=73, Kiran=96}
```

As you can see, the value associated with the key "Harsh" was updated from 65 to 45.

👉Iterating Over a Map

To separate keys and values or iterate through the map, you can use a for-each loop with methods like keySet() to get all keys.

👉Example: Iterating Over Keys and Values

```
import java.util.Map;
import java.util.HashMap;

public class Demo {
    public static void main(String[] args) {
        Map<String, Integer> students = new HashMap<>();
        students.put("Navin", 56);
        students.put("Harsh", 45);
        students.put("Sushil", 73);
        students.put("Kiran", 96);

        for (String name : students.keySet()) {
            System.out.println(name + ": " +
students.get(name));
        }
    }
}
```

👉Output:

```
Navin: 56
Harsh: 45
Sushil: 73
Kiran: 96
```

- The **keySet()** method retrieves only the keys from the map, and we use the get() method to get corresponding values.

👉Important Methods of the Map Interface

Below is a list of commonly used methods in the Map interface along with their descriptions:

Method	Description	Time Complexity
put(Key, Value)	Associates the specified key with the specified value in the map.	O(1)
get(Key)	Returns the value associated with the specified key.	O(1)
containsKey(Key)	Checks if the map contains the specified key. Returns true if it does, otherwise false.	O(1)
containsValue(Value)	Checks if the map contains the specified value. Returns true if it does, otherwise false.	O(n)
isEmpty()	Returns true if the map contains no key-value pairs.	O(1)
clear()	Removes all key-value pairs from the map.	O(n)
remove(Key)	Removes the mapping for a key from this map if it is present.	O(1)
size()	Returns the number of key-value mappings in the map.	O(1)

TELUSKO

Method	Description	Time Complexity
entrySet()	Returns a Set view of the mappings contained in this map.	O(n)
keySet()	Returns a Set view of the keys contained in this map.	O(n)
values()	Returns a collection view of the values contained in this map.	O(n)
putIfAbsent(Key, Value)	Associates the specified value with the specified key if the key is not already associated.	O(1)
replace(Key, Value)	Replaces the value for the specified key with the new value if the key exists.	O(1)
replace(Key, OldValue, NewValue)	Replaces the value for the specified key only if it is currently mapped to the old value.	O(1)

👉Hashtable vs. HashMap

- **HashMap**: Allows null keys and values, and is not synchronized (not thread-safe).
- **Hashtable**: Does not allow null keys or values, and is synchronized (thread-safe).

👉Frequently Asked Questions (FAQs)

Q1. What is a Map Interface in Java?

- A Map interface is a collection of key-value pairs where each key is unique, and you can retrieve values using their corresponding keys.

Q2. What are the types of Map Interface implementations in Java?

- The main implementations of the Map interface are HashMap, LinkedHashMap, and TreeMap.

TELUSKO

13.5 Comparator and Comparable

The Java Collections Framework provides several ways to sort collections, making it easier for developers to manage data. Sorting can be performed using built-in methods or by implementing custom sorting logic. Let's explore how sorting is achieved using the Comparator and Comparable interfaces in Java.

👉1. Sorting in Collections

The Collections API simplifies many tasks for developers, including sorting. It provides utility methods for sorting, making the process straightforward and efficient. Let’s start with a simple example of sorting a list of integers:

```
public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = new ArrayList<>();
        // Since Java 7, specifying the generic type for the
implementing class is optional
        nums.add(4);
        nums.add(3);
        nums.add(7);
        nums.add(9);

        System.out.println(nums); // Output: [4, 3, 7, 9]
        Collections.sort(nums);   // Sorting using built-in
Collections class
        System.out.println(nums); // Output: [3, 4, 7, 9]
    }
}
```

Explanation:

The Collections.sort() method sorts the list in ascending order according to the natural ordering of the elements.

👉 2. Collections Class in Java

The Collections class in Java, part of the java.util package, is a utility class that operates on or returns collections. It provides various static methods that can be used to manipulate collections, such as sorting and searching.

Common Method for Sorting:

- sort(List<T> list):

Sorts the specified list into ascending order, according to the natural ordering of its elements.

👉 3. Custom Sorting Using Comparator

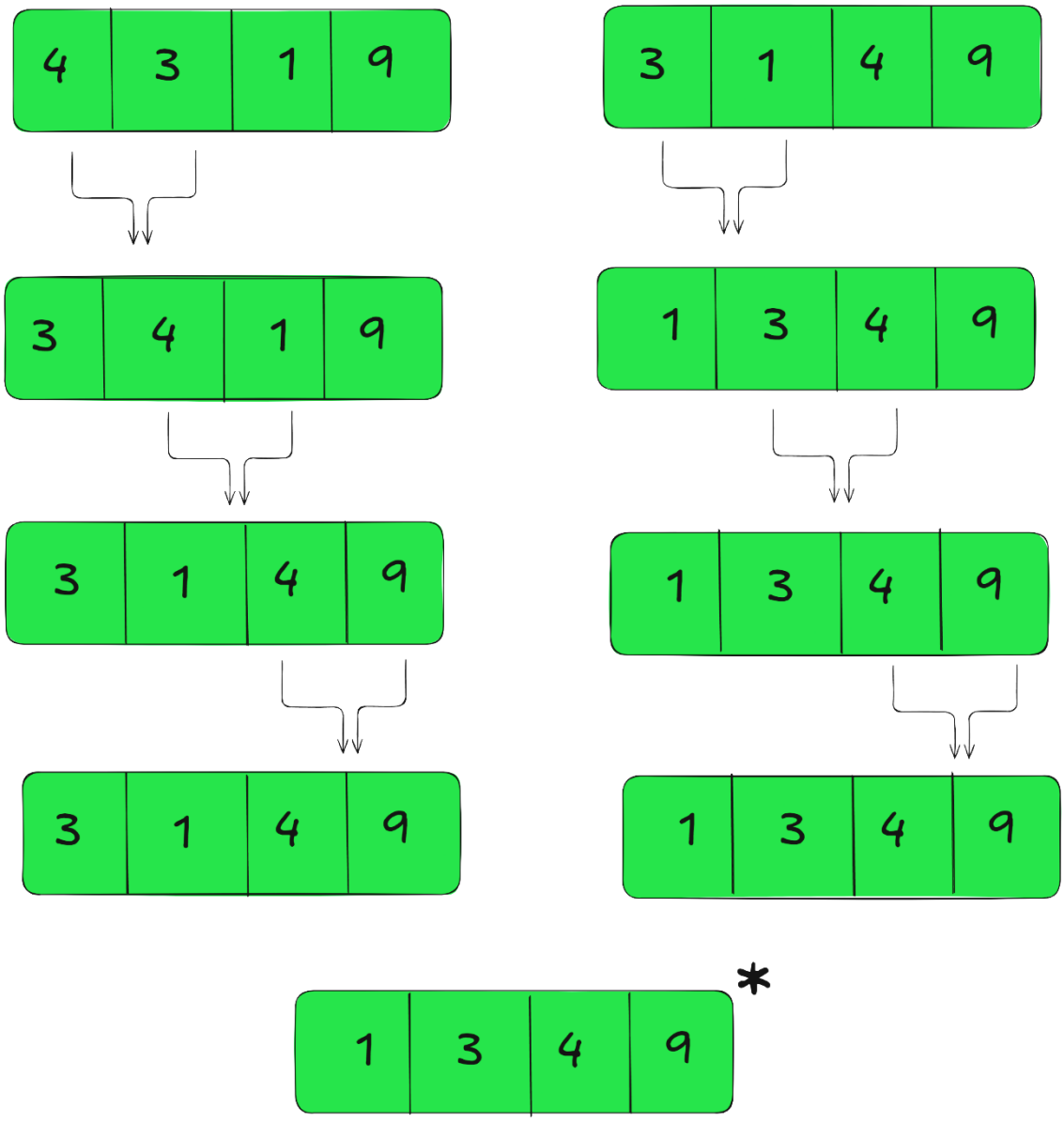
The Comparator interface is used to define a custom sorting order. It is a functional interface, meaning it has a single abstract method compare(), which is implemented to provide sorting logic.

Example of Custom Sorting:

```
public class Demo {
    public static void main(String[] args) {
        // Creating a custom Comparator to sort numbers based
on the last digit
        Comparator<Integer> com = new Comparator<Integer>() {
            public int compare(Integer i, Integer j) {
                if (i % 10 > j % 10)
                    return 1;
                else
                    return -1;
            }
        };

        List<Integer> nums = new ArrayList<>();
        nums.add(54);
        nums.add(33);
        nums.add(41);
        nums.add(69);

        Collections.sort(nums, com); // Sorting using custom
Comparator
        System.out.println(nums);    // Output: [41, 33, 54,
69]
    }
}
```

Explanation:

This example sorts the list based on the last digit of each number. The custom comparator sorts the elements according to the remainder of division by 10.



👉 4. Optimizing with Lambda Expressions

Since Comparator is a functional interface, we can use a lambda expression to simplify the code.

Example Using Lambda:

Comparator<Integer> com = (i, j) -> (i % 10 > j % 10) ? 1 : -1;

The lambda **expression (i, j) -> (i % 10 > j % 10) ? 1 : -1** provides a more concise way to implement the custom sorting logic.

👉 5. Sorting a List of Objects

To sort a list of custom objects (e.g., Student objects), we can use the Comparator interface to specify the sorting criteria.

Example:

```
class Student {
    int age;
    String name;

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class Demo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(33, "Navin"));
        students.add(new Student(12, "John"));
        students.add(new Student(45, "Edward"));
        students.add(new Student(35, "Michael"));

        // Sorting based on age
        Comparator<Student> com = (s1, s2) -> (s1.age >
s2.age) ? 1 : -1;
        Collections.sort(students, com);

        for (Student s : students) {
            System.out.println(s);
        }
```

```
    }
}
```

Output:

```
John (12)
Navin (33)
Michael (35)
Edward (45)
```

Explanation:

This example sorts the list of Student objects based on age. The Comparator sorts the students in ascending order of age.

👉 6. Using the Comparable Interface

The Comparable interface is a part of java.lang package and it provides a way to define the natural ordering of objects. It is a functional interface with a compareTo() method that must be implemented by any class whose objects are intended to be sorted.

Example of Implementing Comparable:

```
class Student implements Comparable<Student> {
    int age;
    String name;

    public Student(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public int compareTo(Student that) {
        if (this.age > that.age)
            return 1;
        else
            return -1;
    }
```

```
    @Override public String toString() {
 return "Student{name='" + name + "', age=" + age + "}"; }
}

public class Demo {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student(33, "Navin"));
        students.add(new Student(12, "John"));
        students.add(new Student(45, "Edward"));
        students.add(new Student(35, "Michael"));

        Collections.sort(students); // Sorting using
Comparable

        for (Student s : students) {
            System.out.println(s);
        }
    }
}
```

Output:

```
Student{name='John', age=12}
Student{name='Navin', age=33}
Student{name='Michael', age=35}
Student{name='Edward', age=45}
```

Explanation:

In this example, the Student class implements Comparable, allowing objects of Student to be sorted directly using Collections.sort().

👉🏻 7. Key Differences Between Comparator and Comparable

- **Comparator:**
  - Used for custom sorting logic.
  - Can sort objects in multiple ways (different comparators for different criteria).
  - Does not require modification of the class whose objects are being sorted.
- **Comparable:**

- Defines the natural ordering of a class.
- Used to compare objects of the same class.
- Requires implementing the compareTo() method in the class.

👉 Conclusion

Both Comparator and Comparable are essential for sorting objects in Java. Comparator is useful for custom sorting, while Comparable provides a natural sorting order for classes. Understanding when to use each interface allows developers to manage sorting tasks efficiently.

Need Of Stream API

👉 Introduction:

Stream API introduced in Java 1.8 version. It is used for **processing collections of data in a functional way**. It allows performing operations like filtering, mapping, and reducing data with **less code** and **better performance**.

Example using a for loop:

1. Suppose we have a list of integers and want to process it step by step.
   - ➢ We initialize a list of integers using Arrays.asList().

```java
import java.util.Arrays;
import java.util.List;

public class Demo {
    public static void main(String[] args) {

        // Step 1: Create a list of integers
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

    }
}
```

2. Filter out only the even numbers and ignore the odd ones.
   - ➢ We iterate through the list and check if each number is even using if (n % 2 == 0).

```java
import java.util.Arrays;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        // Step 1: Create a list of integers
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Step 2: Iterate through the list and filter even numbers
        for (int n : nums) {
            if (n % 2 == 0) {
                // Even number found, process it in the next steps
            }
        }
    }
}
```

3. Double each of the filtered numbers.
   - If the number is even, we multiply it by 2.

```java
import java.util.Arrays;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        // Step 1: Create a list of integers
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Step 2: Iterate through the list and filter even numbers
        for (int n : nums) {
            if (n % 2 == 0) {
                // Step 3: Double the even number
                n = n * 2;
            }
        }
    }
}
```

4. Calculate the sum of all the modified numbers to get the final result.
   - We add the modified numbers to a sum variable and print the final result.

```java
import java.util.Arrays;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        // Step 1: Create a list of integers
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);
        // Step 2: Initialize sum variable
        int sum = 0;
        // Step 3: Iterate through the list, filter even numbers, and modify them
        for (int n : nums) {
            if (n % 2 == 0) {
                // Step 3: Double the even number
                n = n * 2;
            }
            // Step 4: Add the modified number to the sum
            sum += n;
        }
        // Print the final sum
        System.out.println(sum);
    }
}
```

👉 Limitations of for loop & enhanced for loop:

- **Verbose Code** – Requires explicit iteration and condition checks, making the code longer and less readable.
- **Manual Filtering & Transformation** – Needs extra logic for filtering and modifying data, while Streams handle it directly.
- **No Built-in Reduction** – Summing or aggregating values requires manual accumulation.
- **Sequential Execution** – Loops run sequentially by default, whereas Streams support parallel execution easily.
- **No Internal Optimization** – Streams optimize execution with lazy evaluation, but loops process data immediately.

👉 Using Stream API:

```java
import java.util.Arrays;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 2);

        int sum = nums.stream()
            .filter(n -> n % 2 == 0)  // Select even numbers
            .map(n -> n * 2)          // Double the selected numbers
            .reduce(0, Integer::sum); // Sum up the modified values

        System.out.println(sum);
    }
}
```

👉 Explanation:

- **stream()** – Converts the list into a stream for processing.
- **filter(n -> n % 2 == 0)** – Filters out only even numbers.
- **map(n -> n * 2)** – Doubles each filtered number.
- **reduce(0, Integer::sum)** – Accumulates the transformed numbers into a sum.

forEach() method

👉 forEach() in Stream API:

The forEach() method in Java's Stream API is used to iterate over elements in a stream and perform an action on each element.

👉 Key Points:

- **Introduced in Java 8** with the Stream API
- **Purpose**: Iterates through elements and performs an action on each one
- **Takes a Consumer<T>** functional interface as its parameter
- **Processes elements sequentially**, one at a time
- **Consumer<T>** is from the java.util.function package
  - It has a single method: accept(T t)
  - This method performs the operation on each element
- **Best used for**: Side effects like printing, logging, or updating external variables

👉 Example 1: Using Consumer Interface Explicitly

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Creating a Consumer implementation
        Consumer<Integer> con = (Integer n) -> {
            System.out.println(n);
        };

        // Using forEach with the Consumer
        nums.forEach(con);
    }
}
```

Output:

```
4
5
7
3
2
6
```

👉 Example 2: Using Lambda Expression Directly

```java
import java.util.Arrays;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Using forEach with inline lambda
        nums.forEach(n -> System.out.println(n));
    }
}
```

Output:

```
4
5
7
3
2
6
```

👉 Remember:

- forEach() is efficient for iterating over stream elements.
- Ideal for logging, debugging, and performing non-transformational operations.

TELUSKO

Stream API

👉 Stream API:

The Stream API was introduced in Java 8 to provide a functional approach to processing collections. It allows operations like filtering, mapping, and reducing without modifying the original data structure.

👉 Key Characteristics:

- **Stream is an interface** in the java.util.stream package
- Available through the **stream() method** in Collection classes (List, Set, etc.)
- Enables **functional-style operations** on collections
- **Does not modify the original collection**
- **Cannot be reused** after a terminal operation

👉 Stream Pipeline Structure:

1. **Source Stage**
   - Where the stream is created from a data source
   - Example: List.stream()
2. **Intermediate Stage**
   - Operations that transform the data but don't consume it
   - Examples: filter(), map(), sorted()
   - Can be chained together
   - Lazy evaluation (only executed when a terminal operation is called)
3. **Terminal Stage**
   - Operations that consume the stream and close the pipeline
   - Examples: forEach(), collect(), count(), reduce()
   - Once executed, the stream is consumed and closed
   - Attempting to reuse a closed stream will cause IllegalStateException
   - To process the data again, you must create a new stream

TELUSKO

👉 Java Stream API Methods

Methods	Description
filter(Predicate<T> p)	Filters elements based on a condition.
map(Function<T, R> f)	Transforms elements using a function.
sorted()	Sorts elements in natural order.
forEach(Consumer<T> action)	Iterates over elements and performs an action.
reduce(BinaryOperator<T> op)	Reduces stream elements to a single value.
collect(Collector<T, A, R> c)	Converts a stream into a collection or another structure.
findFirst()	Returns the first element in the stream.
toArray()	Converts the stream into an array.

TELUSKO

TELUSKO

Example:

```java
import java.util.Arrays;
import java.util.List;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Long form (step by step)
        // Stream<Integer> s1 = nums.stream();              // Source stage
        // Stream<Integer> s2 = s1.filter(n -> n%2==0);     // Intermediate (even numbers only)
        // Stream<Integer> s3 = s2.map(n -> n*2);           // Intermediate (double each number)
        // int result = s3.reduce(0, (c, e) -> c + e);      // Terminal (sum all numbers)

        // Chained form (same operations)
        int result = nums.stream()                          // Source
                         .filter(n -> n%2==0)               // Keep even numbers: 4, 2, 6
                         .map(n -> n*2)                     // Double each: 8, 4, 12
                         .reduce(0, (c, e) -> c + e);       // Sum: 8 + 4 + 12 = 24

        System.out.println(result);
    }
}
```

Output:

```
24
```

👉 How the Example Works:

- We start with the list: [4, 5, 7, 3, 2, 6]
- filter(n -> n%2==0) keeps only even numbers: [4, 2, 6]
- map(n -> n*2) doubles each number: [8, 4, 12]
- reduce(0, (c, e) -> c + e) sums all numbers with starting value 0: 0 + 8 + 4 + 12 = 24

👉 Remember:

- Once a terminal operation is performed, you must create a new stream to process the data again
- Streams are designed for functional programming, not imperative programming
- Intermediate operations are lazy (not executed until a terminal operation is called)
- The original collection remains unchanged throughout stream operations

TELUSKO

TELUSKO

Map Filter Reduce Sorted

👉 Filter:

The filter() method in Java Stream API is used to filter elements based on a condition.

👉 Key Points:

- Requires a Predicate<T> functional interface
- Predicate<T> contains the test(T t) method that returns a boolean
- Only elements that return true pass through the filter
- We can use lambda expressions to implement the Predicate

Example:

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Without lambda
        Predicate<Integer> p = new Predicate<Integer>() {
            public boolean test(Integer n) {
                return n % 2 == 0;
            }
        };

        // With lambda
        nums.stream()
            .filter(n -> n % 2 == 0)  // Keep only even numbers: 4, 2, 6
            .forEach(n -> System.out.println(n));
    }
}
```

Output:

```
4
2
6
```

TELUSKO

👉 Map:

The **map()** method in Java Stream API is used to **transform each element** of a stream using a given function.

👉 Key Points:

- Takes a Function<T, R> functional interface
- Function<T, R> contains the apply(T t) method
- Transforms each element and returns a new stream
- Commonly used for converting or modifying elements

Example:

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Without lambda
        Function<Integer, Integer> fun = new Function<Integer, Integer>() {
            public Integer apply(Integer n) {
                return n * 2;
            }
        };

        // With lambda
        nums.stream()
            .map(n -> n * 2)  // Double each number: 8, 10, 14, 6, 4, 12
            .forEach(n -> System.out.println(n));
    }
}
```

Output:

```
8
10
14
6
4
12
```

👉 Reduce:

The reduce() method in Java Stream API is used to aggregate elements of a stream into a single result using an accumulation function.

👉 Key Points:

- Used for operations like sum, multiplication, concatenation
- **Three overloaded versions:**

3.1. reduce(T identity, BinaryOperator<T> accumulator)

- identity: Initial/default value
- accumulator: Combines two elements into one
- Returns a value of type T
- Example: reduce(0, (a, b) -> a + b)

3.2. reduce(BinaryOperator<T> accumulator)

- No identity value provided
- Returns an Optional<T> (may be empty if stream is empty)
- Example: reduce((a, b) -> a + b)

3.3. reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)

- Used primarily for parallel streams
- combiner: Merges results from different threads
- Example: reduce(0, (sum, item) -> sum + item, (sum1, sum2) -> sum1 + sum2)

Example:

```java
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Version 1: With identity value
        int sum = nums.stream().reduce(0, (c, e) -> c + e);
        System.out.println("Sum of list: " + sum);

        // Version 2: Without identity value
        Optional<Integer> product = nums.stream().reduce((a, b) -> a * b);
        product.ifPresent(val -> System.out.println("Product: " + val));

        // Version 3: With combiner (for parallel streams)
        int parallelSum = nums.parallelStream()
                              .reduce(0,
                                      (subtotal, element) -> subtotal + element,
                                      (subtotal1, subtotal2) -> subtotal1 + subtotal2);
        System.out.println("Parallel sum: " + parallelSum);
    }
}
```

Output:

```
Sum of list: 27
Product: 5040
Parallel sum: 27
```

👉 Sorted:

The sorted() method in Java Stream API is used to sort elements in a stream either by natural order or using a custom comparator.

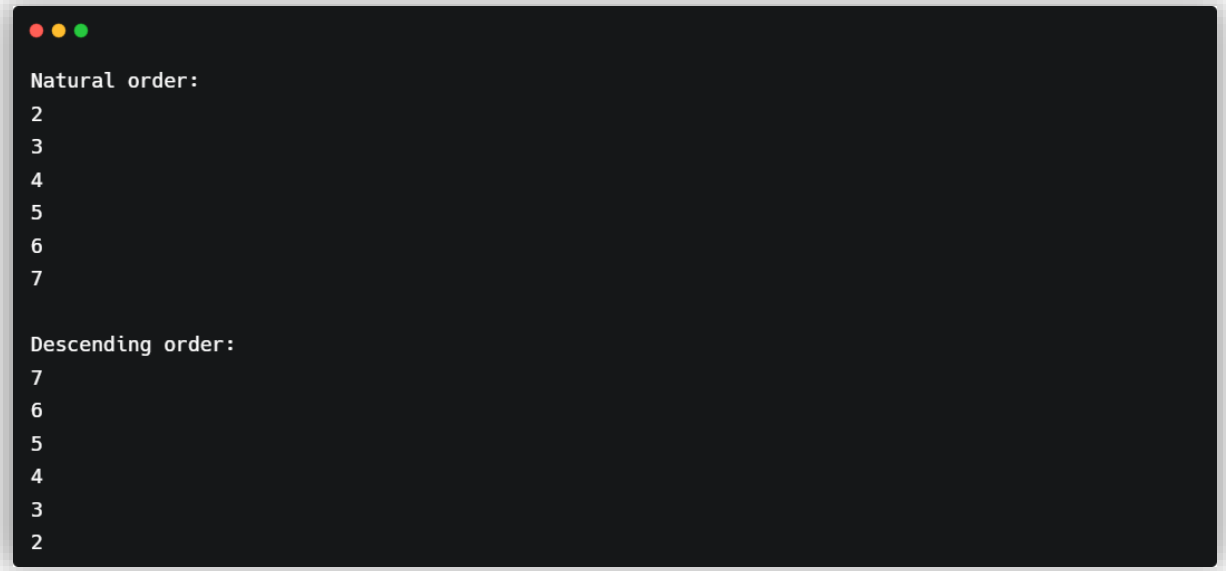
👉 Key Points:

- **Used to sort elements** of a stream in **natural order** or based on a custom comparator.
- **Two variations:**
  1. sorted() – Sorts elements in **natural order** (Comparable<T> must be implemented).
  2. sorted(Comparator<T> c) – Sorts elements based on a **custom comparator**.
- **Returns a new sorted stream**; does not modify the original collection.
- **Efficient for finite streams**, but should be avoided on **infinite streams** due to performance issues.

Example:

```java
import java.util.Arrays;
import java.util.List;
import java.util.Comparator;

public class Demo {
    public static void main(String[] args) {
        List<Integer> nums = Arrays.asList(4, 5, 7, 3, 2, 6);

        // Natural order sorting
        System.out.println("Natural order:");
        nums.stream()
            .sorted()
            .forEach(n -> System.out.println(n));

        // Custom sorting (descending)
        System.out.println("\nDescending order:");
        nums.stream()
            .sorted(Comparator.reverseOrder())
            .forEach(n -> System.out.println(n));
    }
}
```

Output:



When working with **Streams** in Java, we can also leverage **multiple threads** to improve performance.

To do this, Java provides a method called:

👉 **parallelStream()**

- This method allows the **stream operations** to be executed in **parallel**
- It uses **multiple threads** internally to **speed up** processing, especially for large collections.

TELUSKO

Parallel Stream

👉 Parallel Stream:

A Parallel Stream in Java is a type of stream that processes elements concurrently using multiple threads, leveraging multi-core processors to improve performance. It is part of the Java Stream API and is built on the Fork/Join framework.

- **Divides the stream into multiple parts** and processes them simultaneously.
- **Uses multiple threads** to execute operations in parallel.
- **Can be created using:**
  1. collection.parallelStream() – Creates a parallel stream from a collection.
  2. stream.parallel() – Converts a sequential stream into a parallel stream.
- **Order of execution is not guaranteed** due to concurrent processing.
- **May improve performance** for large datasets but may have overhead for small collections.
- **Not thread-safe for shared mutable state**, requiring synchronization in such cases.

👉 When to Use Parallel Streams?

- When working with **large datasets** for better performance.
- When **operations are independent** and do not require ordering.
- When **CPU-intensive tasks** need to be distributed across multiple cores.

Note:

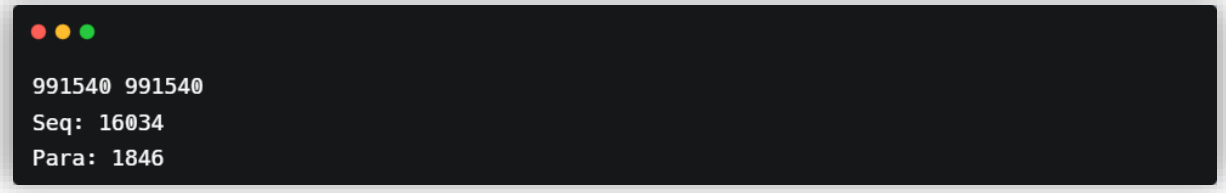
- Avoid parallel stream when processing small datasets.
- Avoid parallel when order of execution is important .
- Avoid parallel when working with shared mutable state, as parallel execution may cause race

TELUSKO

Example:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Demo {
    public static void main(String[] args) {
        int size = 10_000; // Define the size of the list
        List<Integer> nums = new ArrayList<>();
        Random ran = new Random();

        // Populate the list with random integers between 0 and 99
        for (int i = 1; i <= size; i++) {
            nums.add(ran.nextInt(100));
        }

        // Measure execution time for sequential stream processing
        long startSeq = System.currentTimeMillis();
        int sumSeq = nums.stream()
                .map(i -> {
                    try {
                        Thread.sleep(1); // Simulate processing delay
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    return i * 2;
                })
                .mapToInt(i -> i) // Convert Stream<Integer> to IntStream
                .sum();
        long endSeq = System.currentTimeMillis();

        // Measure execution time for parallel stream processing
        long startPara = System.currentTimeMillis();
        int sumPara = nums.parallelStream()
                .map(i -> {
                    try {
                        Thread.sleep(1); // Simulate processing delay
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    return i * 2;
                })
                .mapToInt(i -> i) // Convert Stream<Integer> to IntStream
                .sum();
        long endPara = System.currentTimeMillis();

        // Print results
        System.out.println("Sequential Sum: " + sumSeq + " | Parallel Sum: " + sumPara);
        System.out.println("Sequential Time: " + (endSeq - startSeq) + "ms");
        System.out.println("Parallel Time: " + (endPara - startPara) + "ms");
    }
}
```

Output:



Note:

The output will be depend on system performance.

TELUSKO

TELUSKO

Optional Class

👉 What is Optional in Java?

- Optional is a container object introduced in Java 8.
- It is used to represent the presence or absence of a value.
- Its main goal is to avoid the classic runtime error: NullPointerException.

👉 Why was Optional introduced?

- To handle null values in a safe and clean way without writing too many null checks manually.

👉 Key Points of Optional in Stream API

- It is returned by stream terminal operations like:
  - findFirst()
  - findAny()
  - max() and min()

- Optional helps you:
  - Avoid null pointer exceptions
  - Write cleaner and safer code
  - Provide default values using orElse()
  - Chain fallback logic using orElseGet()
  - Throw custom exception using orElseThrow()

👉 Example of the Optional class:

- Suppose we have a list of names

```java
import java.util.Arrays;
import java.util.List;

public class OptionalEx {
    public static void main(String[] args) {

        // Creating a list of names
        List<String> names = Arrays.asList("Navin", "Laxmi", "John", "Kishor");

    }
}
```

TELUSKO

- We want to find the first name that contains "x"

```java
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class OptionalEx {
    public static void main(String[] args) {

        // Creating a list of names
        List<String> names = Arrays.asList("Navin", "Laxmi", "John", "Kishor");

        // Using Optional to find the first name containing 'x'
        Optional<String> name = names.stream()
                                .filter(str -> str.contains("x"))
                                .findFirst();

        //Printing the found name
        System.out.println(name.get());
    }
}
```

Output:

```
Laxmi
```

- Suppose no name in the list contains the letter “x”, then stream returns an empty Optional, and calling .get() on an empty Optional causes NoSuchElementException.

```java
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class OptionalEx {
    public static void main(String[] args) {

        // Creating a list of names (Updated: "Laxmi" changed to "Lakshmi")
        List<String> names = Arrays.asList("Navin", "Lakshmi", "John", "Kishor");

        // Using Optional to find the first name containing 'x'
        Optional<String> name = names.stream()
                                .filter(str -> str.contains("x"))
                                .findFirst();

        // Printing the found name, if present (May throw NoSuchElementException if not found)
        System.out.println(name.get());
    }
}
```

Output:

```
Exception in thread "main" java.util.NoSuchElementException: No value present
        at java.base/java.util.Optional.get(Optional.java:143)
        at OptionalEx.main(OptionalEx.java:14)
```

- Avoiding NullPointerException using Optional.

```java
import java.util.Arrays;
import java.util.List;

public class OptionalEx {
    public static void main(String[] args) {

        // Creating a list of names
        List<String> names = Arrays.asList("Navin", "Lakshmi", "John", "Kishor");

        // Using Optional to find the first name containing 'x'
        // Optional<String> name = names.stream()
        //                      .filter(str -> str.contains("x"))
        //                      .findFirst();

        // Printing the result using Optional's orElse() to handle absence of match
        // System.out.println(name.orElse("Not Found"));

        // Alternative approach: Directly assigning the result to a String variable
        // Using orElse() to return "Not Found" if no match is found
        String name2 = names.stream()
                            .filter(str -> str.contains("x"))
                            .findFirst()
                            .orElse("Not Found");

        // Printing the result
        System.out.println(name2);
    }
}
```

Output:

```
Not Found
```

👉 Remember

- Optional is a powerful and safe way to handle possibly null results from Stream operations.
- Avoids NullPointerException and makes your code cleaner and safer.
- Always prefer using .orElse() or .orElseGet() instead of .get() directly.

TELUSKO

Method Reference

👉 Method Reference

A method reference is a shorthand notation for a lambda expression that executes just one method. Instead of writing the complete lambda expression with parameters, you can simply refer to an existing method by its name using the **::** (double colon) operator.

👉 Key Points

- Method references use the syntax ClassName::methodName or objectName::methodName
- They provide a more concise way to write lambda expressions that only call a single method
- The **::** (double colon) operator is used to separate the class or object from the method name
- Method references work only when the lambda expression is simply calling another method
- They make your code shorter and often more readable

👉 Method Reference Example

- Let's start with a simple list of names that we'll work with:

```java
import java.util.Arrays;
import java.util.List;

public class MethodRefEx {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Navin", "Harsh", "John");
        System.out.println("Original names: " + names);
    }
}
```

Output:

```
Original names: [Navin, Harsh, John]
```

TELUSKO

- Now, let's use the Stream API with a lambda expression to convert all names to uppercase:

```java
import java.util.Arrays;
import java.util.List;

public class MethodRefEx {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Navin", "Harsh", "John");
        System.out.println("Original names: " + names);

        // Convert to uppercase using lambda expression
        List<String> uNames = names.stream()
                                   .map(name -> name.toUpperCase())
                                   .toList();

        System.out.println("Uppercase names (using lambda): " + uNames);
    }
}
```

Output:

```
Original names: [Navin, Harsh, John]
Uppercase names (using lambda): [NAVIN, HARSH, JOHN]
```

- We can simplify the code above by using a method reference:

```java
import java.util.Arrays;
import java.util.List;

public class MethodRefEx {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Navin", "Harsh", "John");
        System.out.println("Original names: " + names);

        // Convert to uppercase using method reference
        List<String> uNames = names.stream()
                                   .map(String::toUpperCase)
                                   .toList();

        System.out.println("Uppercase names (using method reference): " + uNames);

        // Using method reference with forEach
        System.out.print("Printing each name: ");
        uNames.forEach(System.out::println);
    }
}
```

Output:

```
Original names: [Navin, Harsh, John]
Uppercase names (using method reference): [NAVIN, HARSH, JOHN]
Printing each name: NAVIN
HARSH
JOHN
```

Explanation:

- name -> name.toUpperCase() becomes String::toUpperCase
  - Here, we're saying "call the toUpperCase method on the String object you receive"
- name -> System.out.println(name) becomes System.out::println
  - Here, we're saying "call the println method of System.out with the object you receive"

This makes our code shorter and often easier to read, especially once you get familiar with the syntax.

👉 Types of Method References

1. **Static method reference**: ClassName::staticMethodName
2. **Instance method reference of a particular object**:
   objectName::instanceMethodName
3. **Instance method reference of an arbitrary object of a particular type**:
   ClassName::instanceMethodName
4. **Constructor reference**: ClassName::new

TELUSKO

Constructor Reference

👉 Constructor Reference

A constructor reference is a special kind of method reference that refers to a class constructor. Just like method references, constructor references provide a shorthand way to write lambda expressions, but specifically for creating new objects. They use the same **::** (double colon) operator syntax followed by the keyword new.

👉 Key Points

- Constructor references use the syntax ClassName::new
- They are used to create new objects in functional programming contexts
- Constructor references work like factory methods for creating objects
- Java automatically selects the appropriate constructor based on the context
- They can be used with any constructor (default, parameterized, etc.)
- Constructor references make object creation code more concise in streams

👉 Constructor Reference Example

➢ First, let's create a Student class and a list of names that we'll work with:

```java
import java.util.Arrays;
import java.util.List;

class Student {
    private String name;
    private Integer age;

    // 0-params constructor
    public Student() {
    }

    // Single parameter constructor
    public Student(String name) {
        this.name = name;
    }

    // Getters and Setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    // toString method for printing Student objects
    @Override
    public String toString() {
        return "Student{name='" + name + "', age=" + age + "}";
    }
}

public class ConstructorRefEx {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Navin", "Harsh", "John");

    }
}
```

➢ Now, let's create a Student object for each name using a traditional for loop:

```java
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

// Student class as defined above

public class ConstructorReferenceDemo {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Navin", "Harsh", "John");
        System.out.println("Names: " + names);

        // Create Student objects using a for loop
        List<Student> students = new ArrayList<>();

        for (String name : names) {
            students.add(new Student(name));
        }

        System.out.println("Students (using for loop): " + students);
    }
}
```

Output:

```
Names: [Navin, Harsh, John]
Students (using for loop): [Student{name='Navin', age=0}, Student{name='Harsh', age=0}, Student{name='John', age=0}]
```

- Let's improve our approach by using Stream API with a lambda expression:

```java
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

// Student class as defined above

public class ConstructorReferenceDemo {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Navin", "Harsh", "John");
        System.out.println("Names: " + names);

        // Create Student objects using Stream API with lambda
        List<Student> students = names.stream()
                                      .map(name -> new Student(name))
                                      .toList();

        System.out.println("Students (using stream with lambda): " + students);
    }
}
```

Output:

```
Names: [Navin, Harsh, John]
Students (using stream with lambda): [Student{name='Navin', age=0}, Student{name='Harsh', age=0},
Student{name='John', age=0}]
```

- Finally, we can simplify the code further by using a constructor reference:

```
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

// Student class as defined above

public class ConstructorReferenceDemo {
    public static void main(String[] args) {
        // Create a list of names
        List<String> names = Arrays.asList("Navin", "Harsh", "John");
        System.out.println("Names: " + names);

        // Create Student objects using Stream API with constructor reference
        List<Student> students = names.stream()
                                      .map(Student::new)
                                      .toList();

        System.out.println("Students (using constructor reference): " + students);
    }
}
```

Output:

```
Names: [Navin, Harsh, John]
Students (using constructor reference): [Student{name='Navin', age=0}, Student{name='Harsh',
age=0}, Student{name='John', age=0}]
```

👉 Explanation:

- When you use a constructor reference like Student::new, here's what happens:

1. Java recognizes that you want to create new Student objects
2. It looks for a constructor in the Student class that matches the context
3. In our example, the Stream's map function expects a function that takes a String and returns a Student
4. Java automatically matches this with the Student(String name) constructor
5. Each name from the stream is passed to this constructor, creating a new Student object