# Java Unit 4

**Nested Class and Lambda Expressions:** : Nested Class, Understanding the importance of static and non-static nested classes, Local and Anonymous class, Functional Interface, Lambda expressions

**Utility Classes** : Working with Dates

**Exceptions and Assertions** : Exception overview, Exception class hierarchy and exception types, Propagation of exceptions, Using try, catch and finally for exception handling, Usage of throw and throws, handling multiple exceptions using multi-catch, Autoclose resources with try-with resources statement, Creating custom exceptions, Testing invariants by using assertions

## Nested Class and Lambda Expressions

### Nested Class in Java

A **nested class** in Java is a class defined within another class. It is used to logically group classes that are only used in one place, improving **encapsulation**, **readability**, and **maintainability**. Nested classes can also access private members of the outer class.

**Syntax:**

```
class OuterClass                                        Ummed Singh
{
...
   class NestedClass
   {
        ...
   }
}
```

---

## Types of Nested Classes

Java provides two types of nested classes:

1. **Static Nested Class**
2. **Non-static Nested Class (Inner Class)**

## 1. Static Nested Class

- Defined with the `static` keyword.
- Does not have access to the instance members of the outer class.
- Can only access `static` members of the outer class.
- Can be instantiated without creating an instance of the outer class.

**Example of Static Nested Class**

```
class Outer {                                                    Ummed Singh
    static String outerStaticVar = "Static Variable";

    // Static Nested Class
    static class StaticNested {
        void display() {
            System.out.println("Accessing outer class static
variable: " + outerStaticVar);
        }
    }
}

public class NestedStaticDemo {
    public static void main(String[] args) {
        // Instantiating static nested class without creating
outer class object
        Outer.StaticNested nestedObj = new Outer.StaticNested();
        nestedObj.display();
    }
}
Output:
Accessing outer class static variable: Static Variable
```

## Uses of Static Nested Class

✔ **Memory Efficiency**: Since it doesn't hold a reference to the outer class instance, it reduces memory usage and avoids unintentional memory leaks.

✔ **Independent Behavior**: Can be used even if no instance of the outer class exists, making it useful for utility or helper classes.

✔ **Encapsulation & Organization**:

- Helps in logically grouping related code while keeping it within a single outer class.
- Useful when the nested class should not depend on an instance of the outer class.

✔ **Performance Optimization**: Since it doesn't require an instance of the outer class, it reduces overhead.

---

## 2. Non-static Nested Class (Inner Class)

An **inner class** is a non-static nested class and has access to all members of the outer class, including private members.

### Types of Inner Classes

1. **Member Inner Class:** inside a class but outside of any method
2. **Local Inner Class:** inside a method
3. **Anonymous Inner Class**

### 2.1. Member Inner Class

- Defined inside a class but outside a method.
- Can access all members (including private) of the outer class.
- Requires an instance of the outer class to create an instance of the inner class.

### Example of Member Inner Class

```
class OuterExample {                                    Ummed Singh
    private String message = "Hello from Outer class";

    // Member Inner Class
    class Inner {
        void display() {
            System.out.println("Message from Outer: " + message);
        }
    }
}

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

```
        // Creating an instance of Outer class
        OuterExample outerObj = new OuterExample();

        // Creating an instance of Inner class
        OuterExample.Inner innerObj = outerObj.new Inner();
        innerObj.display();
    }
}
Output:
Message from Outer: Hello from Outer class
```

---

### 2.2. Local Inner Class

- Defined inside a method or block.
- Can access local variables **only if they are `final` or effectively final**.
- Cannot be instantiated outside the method where it's defined.

**Example of Local Inner Class**

```
class OuterDemo {                                          Ummed Singh
    void display() {
        final String localMessage = "Inside Local Inner Class";

        // Local Inner Class
        class LocalInner {
            void show() {
                System.out.println(localMessage);
            }
        }
        // Creating instance of LocalInner inside method
        LocalInner localObj = new LocalInner();
        localObj.show();
    }
}

public class LocalInnerDemo {
    public static void main(String[] args) {
        OuterDemo outerObj = new OuterDemo();
        outerObj.display();
    }
}
```

```
Output:
Inside Local Inner Class
```

---

### 2.3. Anonymous Inner Class

- A class with **no name**, defined **inside a method or as an argument**.
- Used to provide implementation for an interface or an abstract class.
- Cannot have explicit constructors.

**Example of Anonymous Inner Class (Implementing an Interface)**

```
interface Greeting {                                        Ummed Singh
    void sayHello();
}

public class AnonymousDemo {
    public static void main(String[] args) {
        // Anonymous inner class implementing Greeting interface
        Greeting greet = new Greeting() {
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner
Class");
            }
        };

        greet.sayHello();
    }
}
Output:
Hello from anonymous Inner Class
```

## Uses of Non-Static Inner Class

✔ **Encapsulation of Helper Classes**: Used when a nested class is only meaningful within the context of an instance of the outer class.

✔ **Tighter Binding to Outer Class**: Useful when the inner class depends on the outer class's data and methods.

✔ **Encapsulation & Data Hiding**: The inner class can access **private members** of the outer class without exposing them publicly.

✔ **Improved Readability & Maintainability**: Groups related functionality within the outer class rather than creating a separate class file.

---

## Advantages of Nested Classes

1. **Encapsulation** – Improves data hiding by keeping inner details within the outer class.
2. **Code Readability** – Groups related classes together logically.
3. **Better Organization** – Helps in reducing clutter and improving maintainability.
4. **Access to Private Members** – Inner classes can directly access private members of the outer class.

---

## When to Use Nested Classes?

- When a class is **only used within another class**.
- When you want to **improve encapsulation**.
- When you want to **logically group related classes**.
- When a class needs **access to private members of the outer class**.

---

| Nested Class Type | `static` Keyword | Can Access Outer Class Members | Instantiation Requirement |
|---|---|---|---|
| **Static Nested Class** | ✅ Yes | ❌ Only static members | Can be created without outer class instance |
| **Member Inner Class** | ❌ No | ✅ Yes | Requires an outer class instance |
| **Local Inner Class** | ❌ No | ✅ Yes (only final/effectively final local variables) | Only accessible inside the method |
| **Anonymous Inner Class** | ❌ No | ✅ Yes | No explicit name, created inline |

# Functional Interface

A **Functional Interface** in Java is an interface that contains exactly **one abstract method** but may have multiple **default** or **static** methods. It is used in **lambda expressions** and **functional programming** in Java. Functional interfaces were introduced in **Java 8** to support the use of **Lambda Expressions**.

---

## Key Characteristics of Functional Interfaces

1. **Single Abstract Method (SAM)**:
   - A functional interface must have exactly **one abstract method**.
   - This is why they are sometimes called **SAM Interfaces** (Single Abstract Method Interfaces).
2. **Can Have Default and Static Methods**:
   - A functional interface **can have multiple default and static methods**, but it must have only one abstract method.
3. **Works with Lambda Expressions**:
   - Functional interfaces enable the use of **lambda expressions** to provide a concise way to implement their abstract method.
4. ***@FunctionalInterface* Annotation** *(Optional but Recommended)*:
   - This annotation ensures that an interface is truly functional by enforcing that it has only one abstract method.
   - If another abstract method is added by mistake, the compiler will generate an error.

---

# Example of a Functional Interface

```
@FunctionalInterface                                    Ummed Singh

interface MyFunctionalInterface {

   void myMethod(); // Single Abstract Method (SAM)

   // Default method (Optional)

   default void defaultMethod() {
```

```
        System.out.println("Default method in Functional
Interface");

    }

    // Static method (Optional)

    static void staticMethod() {

        System.out.println("Static method in Functional
Interface");

    }

}

public class FunctionalDemo {

    public static void main(String[] args) {

        // Using Lambda Expression to implement Functional
Interface

        MyFunctionalInterface obj = () ->
System.out.println("Implementation using Lambda!");

        obj.myMethod();

        obj.defaultMethod(); // Calling default method

        MyFunctionalInterface.staticMethod(); // Calling static
method

    }

}

Output:
Implementation using Lambda!
Default method in Functional Interface
Static method in Functional Interface
```

## Built-in Functional Interfaces in Java (Java 8)

Java provides several **predefined functional interfaces** in the `java.util.function` package. Some of the most commonly used ones are:

| Interface | Abstract Method | Description |
|---|---|---|
| ***Predicate<T>*** | ***boolean test(T t)*** | Evaluates a condition and returns `true` or `false`. |
| ***Function<T, R>*** | ***R apply(T t)*** | Takes an input of type `T` and returns a result of type `R`. |
| ***Consumer<T>*** | ***void accept(T t)*** | Takes an input but does not return anything. |
| ***Supplier<T>*** | ***T get()*** | Returns a result without taking any input. |
| ***BiFunction<T, U, R>*** | ***R apply(T t, U u)*** | Takes two inputs and produces a result. |

---

# Examples of Built-in Functional Interfaces

### 1. Predicate (Used for Filtering)

A ***Predicate<T>*** is used to test a condition and return ***true*** or ***false***.

```
import java.util.function.Predicate;                    Ummed Singh
public class PredicateExample {
    public static void main(String[] args) {
        // Predicate to check if a number is even
        Predicate<Integer> isEven = num -> num % 2 == 0;

        System.out.println(isEven.test(10)); // Output: true
        System.out.println(isEven.test(15)); // Output: false
    }
}
```

---

## 2. Function (Takes Input and Returns a Result)

A ***Function<T, R>*** takes an input T and returns a result R.

```
import java.util.function.Function;                    Ummed Singh
public class FunctionExample {
    public static void main(String[] args) {
        // Function to calculate the square of a number
        Function<Integer, Integer> square = num -> num * num;

        System.out.println(square.apply(5)); // Output: 25
        System.out.println(square.apply(7)); // Output: 49
    }
}
```

---

## 3. Consumer (Used for Processing, No Return)

A ***Consumer<T>*** takes an input but does not return anything (used for processing data).

```
import java.util.function.Consumer;                    Ummed Singh
public class ConsumerExample {
    public static void main(String[] args) {
        // Consumer to print a message
        Consumer<String> printMessage = message ->
System.out.println("Message: " + message);

        printMessage.accept("Hello, Java!"); // Output: Message:
Hello, Java!
    }
}
```

---

## 4. Supplier (Returns a Value Without Taking Input)

A ***Supplier<T>*** provides a value without taking any input.

```
import java.util.function.Supplier;                    Ummed Singh
public class SupplierExample {
    public static void main(String[] args) {
```

```
        // Supplier to generate a random number
        Supplier<Double> randomValue = () -> Math.random();

        System.out.println(randomValue.get()); // Output: Random
number
    }
}
```

---

## Uses of Functional Interfaces

✔ **Improves Code Readability**: Instead of writing anonymous classes, lambda expressions make the code shorter and more readable.

✔ **Encourages Functional Programming**: Java 8 introduced functional programming concepts, and functional interfaces help achieve this.

✔ **Supports Streams API and Method References**: Functional interfaces are widely used in the Java **Streams API** for operations like filtering, mapping, and reducing.

✔ **Allows Behavior Parameterization**: We can pass behavior (in the form of lambda expressions) as arguments to methods, making the code more flexible.

---

## Comparison: Functional Interface vs. Normal Interface

| Feature | Functional Interface | Normal Interface |
|---|---|---|
| **Number of Abstract Methods** | Exactly **one** | One or more |
| **Can Have Default/Static Methods?** | ✅ Yes | ✅ Yes |
| **Supports Lambda Expressions?** | ✅ Yes | ❌ No |
| **Common Use Case** | Functional Programming (Java 8 and later) | Object-Oriented Design |

---

# Lambda Expression

A **Lambda Expression** in Java is essentially an anonymous method (function) that can be used to implement **functional interfaces**. It eliminates the need for writing long anonymous classes.

Lambda expressions were introduced in **Java 8** to provide a **concise way of representing anonymous functions** (functions that don't have a name and don't belong to a specific class). They help in writing cleaner, more readable, and functional-style code.

### Syntax of Lambda Expression

```
(parameters) -> { body }
```

- **Parameters:** The input arguments (optional if not required).
- **Arrow (->) Operator:** Separates parameters from the body.
- **Body:** Contains the function logic.

---

# Example Without Lambda (Using Anonymous Class)

Before Java 8, we used **anonymous inner classes** to implement interfaces with a single abstract method.

```
// Functional Interface
interface MyFunctionalInterface {                    Ummed Singh
    void show();
}

public class WithoutLambdaExample {
    public static void main(String[] args) {
        MyFunctionalInterface obj = new MyFunctionalInterface() {
            @Override
            public void show() {
                System.out.println("Hello from Anonymous
Class!");
            }
        };
        obj.show();
    }
}
Output:
```

```
Hello from Anonymous Class!
```

---

## Example Using Lambda Expression

Using lambda expressions, we can simplify the above code.

```
public class LambdaExample {                                    Ummed Singh
   public static void main(String[] args) {
       MyFunctionalInterface obj = () ->
System.out.println("Hello from Lambda!");
       obj.show();
   }
}
Output:
```

---

## Functional Interface (Prerequisite for Lambda Expressions)

A **Functional Interface** is an interface that has only **one abstract method**. Java provides the ***@FunctionalInterface*** annotation to enforce this.

Example:

```
@FunctionalInterface                                            Ummed Singh
interface FunctionalInterfaceDemo {
   void show();
}
```

Some predefined functional interfaces in **java.util.function** package:

- ***Consumer<T>*** (accepts one argument, returns nothing)
- ***Supplier<T>*** (returns a value, takes no argument)
- ***Function<T, R>*** (takes an argument, returns a value)
- ***Predicate<T>*** (takes an argument, returns ***boolean***)

# Types of Lambda Expressions

### a) No Parameters

```
interface Greetings {                                          Ummed Singh
    void sayHello();
}

public class LambdaDemoNoPar {
    public static void main(String[] args) {
        Greetings g = () -> System.out.println("Hello, Lambda!");
        g.sayHello();
    }
}
```

### b) With One Parameter

```
interface Printer {                                            Ummed Singh
    void print(String message);
}

public class LambdaWithOnePar {
    public static void main(String[] args) {
        Printer p = message -> System.out.println("Message: " +
message);
        p.print("Hello, World!");
    }
}
```

### c) With Multiple Parameters

```
interface MathOperation {                                      Ummed Singh
    int operate(int a, int b);
}

public class LambdaWithMulPar {
    public static void main(String[] args) {
        MathOperation add = (a, b) -> a + b;
        System.out.println("Sum: " + add.operate(5, 3));
```

```
    }
}
```

---

## Using Lambda in Collections (Sorting Example)

### Before Java 8 (Anonymous Class):

```
import java.util.*;                                          Ummed Singh

public class SortingWithAnonymous {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Java", "Python",
"C");

        // Before Java 8
        Collections.sort(names, new Comparator<String>() {
            public int compare(String s1, String s2) {
                return s1.compareTo(s2);
            }
        });

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

### Using Lambda Expression:

```
import java.util.*;                                          Ummed Singh

public class SortingWithLambda {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Java", "Python",
"C");

        // Using Lambda
        Collections.sort(names, (s1, s2) -> s1.compareTo(s2));
```

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

---

## Advantages of Lambda Expressions

✅ **Concise Code** – Reduces boilerplate code
✅ **Improved Readability** – Easier to understand
✅ **Encourages Functional Programming** – Helps in writing declarative code
✅ **Better Performance** – Eliminates the overhead of anonymous classes

---

## Limitations of Lambda Expressions

❌ **Difficult to Debug** – Since lambda expressions have no name, debugging can be tricky.
❌ **Limited Functionality** – Can only be used with functional interfaces (single abstract method).
❌ **Not Suitable for Complex Logic** – If the lambda body is too long, it affects readability.

---

# Utility Classes

## Utility Classes

Utility classes in Java are classes that contain only **static methods** and are used to provide common, reusable functionality across multiple parts of an application. These classes typically do not require instantiation and are designed to be used directly by calling their static methods.

## `java.util` Package

The `java.util` package in Java is one of the most widely used packages as it provides a collection of utility classes and interfaces for data structures, date/time manipulation, random number generation, and more.

## Categories of Classes in `java.util`

The `java.util` package includes:

1. **Collection Framework** – Interfaces and classes for data structures like `List`, `Set`, `Map`, `Queue`, etc..
2. **Utility Classes** – Helper classes like `Objects`, `Optional`, `Properties`, `Random`.
3. **Concurrency Utilities** – Classes like `Timer`, `TimerTask` for scheduling tasks.
4. **Legacy Data Structures** – Older classes like `Vector`, `Stack`, `Hashtable` (mostly replaced by modern collection classes).
5. **Scanner and Formatter** – Used for input parsing and output formatting.
6. **Date and Time Utilities** – Classes for handling dates and times (`Date`, `Calendar`, `TimeZone`)

---

## Important Classes and Interfaces in `java.util`

### A) Collection Framework

The Collection Framework provides various data structures like lists, sets, and maps.

#### 1. Interfaces in Collection Framework

| Interface | Description |
|---|---|
| `Collection<E>` | Root interface for all collection classes. |
| `List<E>` | Ordered collection (e.g., `ArrayList`, `LinkedList`). |
| `Set<E>` | Unordered collection with unique elements (`HashSet`, `TreeSet`). |
| `Queue<E>` | FIFO (First In, First Out) collection (`PriorityQueue`, `LinkedList`). |
| `Deque<E>` | Double-ended queue (`ArrayDeque`, `LinkedList`). |
| `Map<K, V>` | Key-value pairs (`HashMap`, `TreeMap`). |

#### 2. Classes in Collection Framework

| Class | Description |
|---|---|
| `ArrayList<E>` | Dynamic array implementation of `List`. |
| `LinkedList<E>` | Doubly linked list implementation of `List`. |
| `HashSet<E>` | Hash table implementation of `Set`. |
| `TreeSet<E>` | Sorted `Set` implemented using Red-Black tree. |
| `HashMap<K, V>` | Hash table implementation of `Map`. |
| `TreeMap<K, V>` | Sorted `Map` using Red-Black tree. |

**Example: Using ArrayList**

```
public class ArrayListDemo {                                    Ummed Singh
    public static void main(String[] args) {
        ArrayList<String> companies = new ArrayList<>();
        companies.add("Google");
        companies.add("Microsoft");
        companies.add("OpenAI");

        System.out.println(companies.get(1)); // Output:
Microsoft
    }
}
```

---

## B) Utility Classes

The `java.util` package provides several general-purpose utility classes.

| Class | Description |
|---|---|
| `Random` | Generates random numbers. |
| `Objects` | Utility methods for working with objects. |
| `Optional<T>` | Handles optional values, avoiding `null` checks. |
| `Properties` | Handles configuration properties. |

**Example: Using `Random`**

```
import java.util.Random;                                    Ummed Singh
public class RandomDemo {
   public static void main(String[] args) {
       Random random = new Random();
       System.out.println(randome.nextInt(100)); // Random
number between 0 and 99
   }
}
```

---

## C) Concurrency Utilities

The `java.util` package contains classes for scheduling tasks.

| Class | Description |
|---|---|
| `Timer` | Schedules tasks to run at a specific time. |
| `TimerTask` | Defines a task that can be scheduled. |

**Example: Using `Timer`**

```
import java.util.Timer;                                     Ummed Singh
import java.util.TimerTask;
public class TimerDemo {
   public static void main(String[] args) {
       Timer timer = new Timer();
       timer.schedule(new TimerTask() {
           @Override
           public void run() {
               System.out.println("Task executed!");
           }
       }, 2000); // Runs after 2 seconds
   }
}
```

---

## D) Scanner

The `Scanner` class in Java is part of the `java.util` package and is used to take input from the user.

| Class | Description |
|---|---|
| `Scanner` | Reads input from keyboard, files, or streams. |

**Methods of `Scanner` Class**

| Method | Description |
|---|---|
| `nextInt()` | **Read an integer.** |
| `nextDouble()` | **Reads a floating-point number.** |
| `nextBoolean()` | **Reads a boolean (`true`/`false`).** |
| `next()` | **Read a single word (until space).** |
| `nextLine()` | **Read the full line including spaces.** |

**Example: Using `Scanner`**

```
import java.util.Scanner;                                    Ummed Singh
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name);
        scanner.close();
    }
}
```

---

## Utility Classes for Working with Dates

Java provides several utility classes for working with dates and time. The key utility classes are found in the ***java.time*** package (introduced in Java 8) and in the older ***java.util.Date*** and ***java.util.Calendar*** classes. The modern ***java.time*** package is recommended for date and time handling because it is more robust, thread-safe, and feature-rich.

---

# The *java.time* Package (Java 8+)

This package provides several classes for working with dates, times, and durations. The most commonly used classes include:

## (a) *LocalDate* (Date without time)

Represents a date (year, month, day) without time or time zone.

**Example:**

```
public class DateExample {                                  Ummed Singh
    public static void main(String[] args) {
        LocalDate today = LocalDate.now(); // Gets the current
date
        System.out.println("Today's Date: " + today);

        LocalDate specificDate = LocalDate.of(2025, 2, 25); //
Creates a specific date
        System.out.println("Specific Date: " + specificDate);
    }
}
```

## (b) *LocalTime* (Time without date)

Represents time (hour, minute, second, nanosecond) without a date or time zone.

**Example:**

```
import java.time.LocalTime;                                 Ummed Singh
public class TimeExample {
    public static void main(String[] args) {
        LocalTime now = LocalTime.now();
        System.out.println("Current Time: " + now);
```

```
        LocalTime specificTime = LocalTime.of(14, 30, 15);
        System.out.println("Specific Time: " + specificTime);
    }
}
```

### (c) *`LocalDateTime`* (Date and Time)

Represents a combination of date and time without a time zone.

**Example:**

```
import java.time.LocalDateTime;                          Ummed Singh
public class DateTimeExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        System.out.println("Current DateTime: " + now);

        LocalDateTime specificDateTime = LocalDateTime.of(2025,
2, 25, 14, 30);
        System.out.println("Specific DateTime: " +
specificDateTime);
    }
}
```

### (d) *`ZonedDateTime`* (Date, Time, and Time Zone)

Represents a date-time with a time zone.

**Example:**

```
import java.time.ZonedDateTime;                          Ummed Singh
import java.time.ZoneId;
public class ZonedDateTimeExample {
    public static void main(String[] args) {
        ZonedDateTime zdt = ZonedDateTime.now();
        System.out.println("Current ZonedDateTime: " + zdt);

        ZonedDateTime specificZoneTime =
ZonedDateTime.now(ZoneId.of("America/New_York"));
        System.out.println("Time in New York: " +
specificZoneTime);
```

```
    }
}
```

### (e) *`Instant`* (Timestamp)

Represents an instantaneous point on the time scale (useful for timestamps).

**Example:**

```
import java.time.Instant;                                    Ummed Singh
public class InstantExample {
    public static void main(String[] args) {
        Instant now = Instant.now();
        System.out.println("Current Timestamp: " + now);
    }
}
```

---

## Formatting and Parsing Dates (*`DateTimeFormatter`*)

Java provides the ***`DateTimeFormatter`*** class for formatting and parsing date-time objects.

**Example:**

```
import java.time.LocalDateTime;                              Ummed Singh
import java.time.format.DateTimeFormatter;

public class FormattingExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();

        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
        String formattedDate = now.format(formatter);

        System.out.println("Formatted Date: " + formattedDate);
    }
}
```

## Calculations with Dates (Adding and Subtracting)

You can perform date arithmetic using methods like ***.plusDays(), .minusDays()***, etc.

**Example:**

```
import java.time.LocalDate;                                    Ummed Singh

public class DateArithmetic {
    public static void main(String[] args) {
        LocalDate today = LocalDate.now();

        LocalDate nextWeek = today.plusDays(7);
        System.out.println("Date After 7 Days: " + nextWeek);

        LocalDate lastMonth = today.minusMonths(1);
        System.out.println("Date One Month Ago: " + lastMonth);
    }
}
```

## Calculating Differences Between Dates (*Period* and *Duration*)

### Using *Period* for Date Differences

Used for differences in terms of years, months, and days.

**Example:**

```
import java.time.LocalDate;                                    Ummed Singh
import java.time.Period;

public class DateDifference {
    public static void main(String[] args) {
        LocalDate startDate = LocalDate.of(2020, 5, 15);
        LocalDate endDate = LocalDate.of(2025, 2, 25);
```

```
        Period period = Period.between(startDate, endDate);
        System.out.println("Years: " + period.getYears() + ",
Months: " + period.getMonths() + ", Days: " + period.getDays());
    }
}
```

### Using *`Duration`* for Time Differences

Used for differences in hours, minutes, seconds, and nanoseconds.

**Example:**

```
import java.time.Duration;                                  Ummed Singh
import java.time.LocalTime;

public class TimeDifference {
    public static void main(String[] args) {
        LocalTime startTime = LocalTime.of(10, 30);
        LocalTime endTime = LocalTime.of(14, 45);

        Duration duration = Duration.between(startTime, endTime);
        System.out.println("Hours: " + duration.toHours() + ",
Minutes: " + duration.toMinutes());
    }
}
```

---

## The Legacy Date API (*`java.util.Date`* and *`java.util.Calendar`*)

Before Java 8, ***`Date`*** and ***`Calendar`*** were used, but they had several drawbacks like being mutable and not thread-safe.

### Using *`Date`* (Legacy API)

```
import java.util.Date;                                      Ummed Singh

public class OldDateExample {
    public static void main(String[] args) {
        Date date = new Date();
```

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

### Using *Calendar* (Legacy API)

```
import java.util.Calendar;                                  Ummed Singh

public class CalendarExample {
    public static void main(String[] args) {
        Calendar calendar = Calendar.getInstance();
        System.out.println("Current Date: " +
calendar.getTime());

        calendar.add(Calendar.DAY_OF_MONTH, 7);
        System.out.println("Date After 7 Days: " +
calendar.getTime());
    }
}
```

---

## Converting Between Old and New API

Since Java 8, you can convert between ***Date*** and ***LocalDateTime*** easily.

### Convert *Date* to *LocalDateTime*

```
import java.time.Instant;                                   Ummed Singh
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.Date;

public class ConvertDate {
    public static void main(String[] args) {
        Date date = new Date();
        Instant instant = date.toInstant();
        LocalDateTime localDateTime =
instant.atZone(ZoneId.systemDefault()).toLocalDateTime();
```

```
        System.out.println("Converted LocalDateTime: " +
localDateTime);
    }
}
```

### Convert *`LocalDateTime`* to *`Date`*

```
import java.time.LocalDateTime;                          Ummed Singh
import java.time.ZoneId;
import java.util.Date;

public class ConvertToDate {
    public static void main(String[] args) {
        LocalDateTime localDateTime = LocalDateTime.now();
        Date date =
Date.from(localDateTime.atZone(ZoneId.systemDefault()).toInstant
());

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

---

# Exceptions and Assertions

## Exception Overview

#### Exception

An **exception** in Java is an event that **disrupts the normal flow of a program**. It is triggered during runtime when an operation cannot be successfully completed due to an unexpected issue, such as invalid user input, resource unavailability, or incorrect calculations.

When an exception occurs, Java provides a way to handle it gracefully using a mechanism called **exception handling**, ensuring that the application doesn't crash abruptly.

**Why Are Exceptions Important?**

Handling exceptions properly helps in:

- Preventing application crashes.
- Providing meaningful error messages to users.
- Allowing the program to recover and continue execution.
- Improving code maintainability and debugging.

---

## Common Causes of Exceptions

Here are some common scenarios where exceptions occur:

**Invalid User Input**
If a program expects a number but receives a string, it can cause an exception.

**Example:**

```
public static void main(String[] args) {                Ummed Singh
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int num = scanner.nextInt();  // If the user enters a
non-numeric value, it will throw InputMismatchException
    System.out.println("You entered: " + num);
}
```

**Attempting to Access an Array Index Out of Bounds**
If you try to access an index in an array that doesn't exist, an `ArrayIndexOutOfBoundsException` occurs.

**Example:**

```
public static void main(String[] args) {                Ummed Singh
    int[] numbers = {1, 2, 3};
```

```
    System.out.println(numbers[5]); //
ArrayIndexOutOfBoundsException
}
```

**Dividing a Number by Zero**

In Java, division by zero results in an `ArithmeticException`.

**Example:**

```
public static void main(String[] args) {
    int result = 10 / 0;  // Throws ArithmeticException
    System.out.println(result);
}
```

**Opening a Non-Existent File**

If a file is not found when trying to read or write, a `FileNotFoundException` occurs.

**Example:**

```
public static void main(String[] args) {                Ummed Singh
    try {
        File file = new File("nonexistent.txt");
        Scanner scanner = new Scanner(file);
    } catch (FileNotFoundException e) {
        System.out.println("File not found: " + e.getMessage());
    }
}
```

**Network Connectivity Issues**

When trying to connect to a remote server or database, a `SocketException` or `IOException` can occur if the network is unavailable.

**Example:**

```
public static void main(String[] args) {                Ummed Singh
    try {
        URL url = new URL("http://example.com");
        HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
        connection.connect();
    } catch (Exception e) {
        System.out.println("Network issue: " + e.getMessage());
    }
}
```

## How Java Handles Exceptions

When an exception occurs, Java creates an **exception object** containing details about the error (e.g., type of error, message, location in code). This object is passed up the **call stack** until it is caught by an appropriate exception handler (a `try-catch` block).

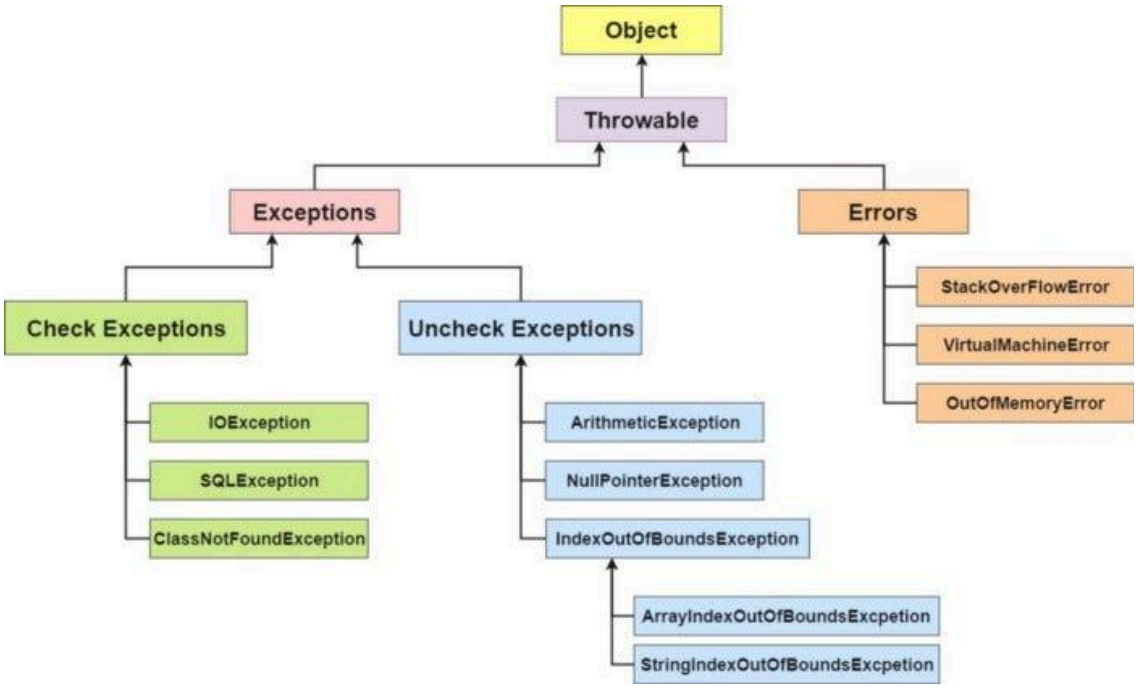
### Exception Lifecycle

1. An error occurs.
2. The Java Runtime Environment (JRE) creates an exception object.
3. The exception propagates up the call stack.
4. If handled (`try-catch`), the program continues execution.
5. If not handled, the program terminates abnormally.

# Exception Class Hierarchy

### Exception Class Hierarchy

In Java, exceptions are represented as objects that inherit from the `Throwable` class. The hierarchy follows this structure:



1. `Throwable` **Class**
    - The root of the exception hierarchy.
    - All exceptions and errors inherit from `Throwable`.
2. `Exception` **Class**
    - Represents exceptional conditions that applications **should** handle.
    - It includes both **checked** and **unchecked** exceptions.
3. `Error` **Class**
    - Represents serious problems that applications **should not** handle.
    - These are caused by the Java Virtual Machine (JVM) and include memory and system failures.

---

**Exception Types**

1. **Checked Exceptions**

    Checked exceptions are exceptions that are checked at compile time. If a method contains code that might throw a checked exception, it must either handle the exception using `try-catch` or declare it using the `throws` keyword.

    - These exceptions must be handled either using `try-catch` or declared using `throws`.
    - Occurs during **compile-time**.
    - Examples:
        - `IOException`: Occurs when a file is not found.
        - `SQLException`: Occurs during database interactions.
        - `ClassNotFoundException`: Thrown when a required class is missing.

    **Example: Exception handling using try-catch block**

```
public static void main(String[] args) {          Ummed Singh

    try {

        FileReader file = new FileReader("test.txt");

    } catch (FileNotFoundException e) {

        System.out.println("File not found: " +
e.getMessage());

    }
```

```
}
```

- The code attempts to read a file that may not exist.
- Since `FileReader` may throw an `IOException`, we must handle it with `try-catch` or declare it using `throws`.

**Example: Using `throws`:**

Instead of handling the exception inside the method, we can declare it using throws.

```
    public class CheckedExceptionWithThrow {

        public static void readFile() throws IOException {

            FileReader fr = new FileReader("demo.txt");

        }

        public static void main(String[] args) {

            try {

                readFile();

            } catch (IOException e) {

                System.out.println("Exception handled: " +
e.getMessage());

            }

        }

    }
```

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

Unchecked exceptions are exceptions that occur at runtime and are not checked at compile time. These exceptions are subclasses of `RuntimeException` and do not require explicit handling.

- These exceptions **do not require** mandatory handling.
- They occur during **runtime** due to programming errors.
- Examples:
  - `NullPointerException`: Trying to access a method or field of a `null` object.
  - `ArithmeticException`: Division by zero.
  - `IndexOutOfBoundsException`: Accessing an invalid array index.

**Example:**

```
public class UncheckedExceptionExample {          Ummed Singh

    public static void main(String[] args) {

        int[] numbers = {1, 2, 3};

        System.out.println(numbers[5]); //
ArrayIndexOutOfBoundsException

    }

}
```

3. **Errors**
   - These indicate serious problems that cannot be handled.
   - Examples:
     - `StackOverflowError`: Infinite recursion.
     - `OutOfMemoryError`: Insufficient heap space.

**Example of StackOverflowError:**

```
public class StackOverflowExample {               Ummed Singh

    static void recursiveMethod() {

        recursiveMethod(); // Infinite recursion

    }
```

```
    public static void main(String[] args) {
        recursiveMethod(); // Causes StackOverflowError
    }
}
```

---

### Key Differences Between Checked and Unchecked Exceptions

| Feature | Checked Exceptions | Unchecked Exceptions |
|---|---|---|
| Occurrence | Compile-time | Runtime |
| Handling Required? | Yes, must be handled with `try-catch` or `throws` | No mandatory handling |
| Examples | `IOException`, `SQLException` | `NullPointerException`, `ArithmeticException` |

## Propagation of Exceptions

#### Exception Propagation

Exception propagation refers to how an exception moves up the call stack when it is not caught in the method where it occurs. If a method throws an exception but does not handle it, the exception propagates to the method that called it. This continues until either:

1. The exception is caught and handled somewhere in the call stack, or

2. It reaches the `main()` method. If `main()` does not handle it, the JVM terminates the program and prints a stack trace.

---

**Example of Exception Propagation**

```
public class ExceptionPropagation {                    Ummed Singh
    static void method1() {
        int result = 10 / 0; // ArithmeticException
    }

    static void method2() {
        method1(); // Exception propagates from method1
    }

    public static void main(String[] args) {
        method2(); // Exception reaches main()
    }
}
```

**Explanation:**

1. `method1()` contains an error (`10 / 0`), which throws an `ArithmeticException`.
2. Since `method1()` does not handle the exception, it propagates to `method2()`, which called `method1()`.
3. `method2()` also does not handle the exception, so it propagates further to `main()`.
4. Since `main()` does not handle it either, the JVM terminates the program and prints the exception message.

**Output:**

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at com.lpu.unit4.exceptions.ExceptionPropagation.method1(ExceptionPropagation.java:5)
    at com.lpu.unit4.exceptions.ExceptionPropagation.method2(ExceptionPropagation.java:9)
    at com.lpu.unit4.exceptions.ExceptionPropagation.main(ExceptionPropagation.java:13)
```

---

## Handling Exception Propagation

To prevent abrupt termination, exceptions can be caught at any level of the call stack using `try-catch`.

**Example: Handling the Exception in `main()`**

```
public class ExceptionHandlingExample {                Ummed Singh
    static void method1() {
        int result = 10 / 0; // ArithmeticException
    }
    static void method2() {
        method1();
    }
    public static void main(String[] args) {
        try {
            method2(); // Exception propagates, but is caught
here
        } catch (ArithmeticException e) {
            System.out.println("Exception handled: " +
e.getMessage());
        }
    }
}

Output:
Exception handled: / by zero
```

---

## Key Points About Exception Propagation

1. **Unchecked exceptions (Runtime exceptions) automatically propagate up the call stack.**
    - Examples: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.
2. **Checked exceptions do NOT propagate unless declared using `throws`.**

    **Example:**

    ```
    class CheckedExceptionPropagation {

       static void method3() throws FileNotFoundException{

           System.out.println("Method 3");
    ```

```
        FileReader file = new FileReader("demo.txt");

        System.out.println("method 3 executed after
exception");

    }

    static void method2() throws FileNotFoundException{

        System.out.println("method 2");

        method3();

        System.out.println("Method 2 executed after call to
method 3");

    }

    static void method1(){

        System.out.println("Method 1");

        try {

            method2();

        }catch (Exception e){

            System.out.println(e.getMessage());

        }

        System.out.println("Method 1 executed after call to
method 2");

    }

    public static void main(String[] args) {

        System.out.println("Main method");

        method1();

        System.out.println("main method executed after call
to method 1");
```

```
    }

}



Output:
Main method
Method 1
method 2
Method 3
demo.txt (The system cannot find the file specified)
Method 1 executed after call to method 2
main method executed after call to method 1
```

3. **If an exception is not caught anywhere, the program terminates abnormally, printing a stack trace.**

---

## Exception Handling Using `try`, `catch`, and `finally`

### 1. `try` Block

- The `try` block contains the code that might throw an exception.
- If an exception occurs inside `try`, it will be thrown and can be caught in a `catch` block.

### 2. `catch` Block

- The `catch` block is used to handle exceptions thrown from the `try` block.
- It can contain custom logic for handling the exception.

### 3. `finally` Block

- The `finally` block is always executed, whether an exception occurs or not.
- It is typically used to release resources like database connections, file streams, etc.

---

### Example: Using `try`, `catch`, and `finally`

```
public class TryCatchFinallyExample {                    Ummed Singh
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // This will cause an
ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " +
e.getMessage());
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
Output:
Exception caught: / by zero
This block always executes.
```

---

## Using `throw` and `throws`

### `throw` Keyword

- The `throw` keyword is used to explicitly throw an exception.
- It is typically used inside a method or a block of code.
- The exception object thrown must be of type `Throwable` or a subclass of it.

**Example of `throw`**

```
public class ThrowExample {                              Ummed Singh
    static void validate(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or
above");
        }
    }
    public static void main(String[] args) {
        validate(16); // Throws IllegalArgumentException
    }
}
```

**Explanation:**

- The `validate` method checks if the `age` is below 18.
- If the condition is met, an `IllegalArgumentException` is thrown.
- Since there is no `try-catch`, the program terminates with an error.

---

## `throws` Keyword

- The `throws` keyword is used to declare exceptions that a method might throw.
- It is placed in the method signature.
- This informs the caller of the method that it must handle or propagate the exception.

**Example of `throws`**

```
public class ThrowsExample {                                    Ummed Singh
    static void readFile() throws IOException {
        FileReader file = new FileReader("nonexistent.txt");
        file.read();
    }
    public static void main(String[] args) {
        try {
            readFile(); // Must be handled because of throws
        } catch (IOException e) {
            System.out.println("Exception handled: " + e);
        }
    }
}
```

**Explanation:**

- The `readFile` method declares that it may throw an `IOException`.
- Since `IOException` is a checked exception, the caller (`main`) must handle it using `try-catch`.

---

## Key Differences Between `throw` and `throws`

| Feature | `throw` | `throws` |
|---|---|---|

| Purpose | Used to throw an exception inside a method | Used in method signature to declare an exception |
|---|---|---|
| Placement | Inside method body | After method signature |
| Type of Exception | Must be an instance of `Throwable` | Can be checked or unchecked exceptions |
| Requires Handling? | No, but must be caught somewhere | Yes, must be handled or propagated |

# Handling Multiple Exceptions Using Multi-Catch

Java allows catching multiple exceptions in a **single** `catch` block using the **multi-catch** feature introduced in **Java 7**. This reduces code duplication and improves readability.

---

## How Multi-Catch Works

- Multiple exceptions can be caught using a **pipe (`|`) operator** in a **single** `catch` block.
- The variable **`e`** will be a common reference for all exceptions listed in the `catch` block.
- The exceptions specified must not have an **inheritance relationship** (i.e., one should not be a subclass of the other).

---

## Example 1: Handling Multiple Exceptions

```
public class MultiCatchExample {                                    Ummed Singh
    public static void main(String[] args) {
        try {
            int[] numbers = new int[5];
            numbers[10] = 50; // Causes
ArrayIndexOutOfBoundsException
            int result = 10 / 0; // Causes ArithmeticException
        } catch (ArithmeticException |
ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e);
        }
    }
```

```
}


Output:
Exception caught: java.lang.ArrayIndexOutOfBoundsException:
Index 10 out of bounds for length 5
```

**Explanation:**

- The `try` block contains **two possible exceptions**:
  - **`ArrayIndexOutOfBoundsException`** when accessing an invalid array index.
  - **`ArithmeticException`** due to division by zero.
- The **first exception** (`ArrayIndexOutOfBoundsException`) occurs, and execution jumps to the `catch` block.
- Since both exceptions are handled in the same block, the program does not crash.

---

## Example 2: Handling Multiple Exceptions without Pipe

```
public class MulthipleCatchDemo {                    Ummed Singh
   public static void main(String[] args) {
       int[] arr = {10, 20, 30};
       try {
           System.out.println(arr[1]); //
ArrayIndexOutOfBoundsException
           int num = 10 / 0; // ArithmeticException
       } catch (IllegalArgumentException e) {
           System.out.println(e.getMessage());
       }catch (ArrayIndexOutOfBoundsException e){
           System.out.println(e.getMessage());
       }
       catch (Exception e) {
           System.out.println("Exception.....");
       }
   }
}
```

## Key Points to Remember

1. **Improves code efficiency** by reducing duplicate `catch` blocks.
2. The **exceptions must be unrelated** (not in the same inheritance hierarchy).
3. The **common exception reference (`e`) cannot call exception-specific methods** because it is a general reference.
4. It helps in **better exception handling** with cleaner code.

---

### When NOT to Use Multi-Catch

You **cannot** use multi-catch if exceptions have an **inheritance relationship**,
**Example:**

```
try {                                                    Ummed Singh
    // Some code

} catch (IOException | FileNotFoundException e) { // Compilation
Error!

    System.out.println("Exception caught: " + e);

}
```

**Why?**

- `FileNotFoundException` is a subclass of `IOException`, so they are related.
- In such cases, **catch only the parent class** (`IOException`) instead.

## Auto-Close Resources with `try-with-resources`

The **`try-with-resources`** statement was introduced in **Java 7** to automatically close resources such as files, database connections, sockets, and streams. This eliminates the need for explicitly closing resources in a `finally` block, reducing the risk of resource leaks.

---

### How `try-with-resources` Works

- Any resource declared inside the `try` parentheses **must implement the `AutoCloseable`** or `Closeable` interface.
- The resource is **automatically closed** when the `try` block finishes execution, regardless of whether an exception occurs or not.

---

## Example of `try-with-resources`

```
public class TryWithResourcesExample {                Ummed Singh
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("test.txt"))) {
            System.out.println(br.readLine()); // Read and print
first line
        } catch (IOException e) {
            System.out.println("Exception handled: " + e);
        }
    }
}
```

**Explanation:**

1. **`BufferedReader` and `FileReader` are declared inside `try`**
    - Since `BufferedReader` implements `AutoCloseable`, it will be **automatically closed** when the `try` block finishes execution.
2. **No need for an explicit `finally` block**
    - Normally, without `try-with-resources`, we would have to write a `finally` block to close `BufferedReader`.
    - Here, Java handles it **implicitly**.

---

## Equivalent Code Without `try-with-resources` (Before Java 7)

```
import java.io.*;                                     Ummed Singh
public class WithoutTryWithResources {
    public static void main(String[] args) {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new FileReader("test.txt"));
            System.out.println(br.readLine());
        } catch (IOException e) {
```

```
            System.out.println("Exception handled: " + e);
        } finally {
            try {
                if (br != null) br.close(); // Manually closing
the resource
            } catch (IOException e) {
                System.out.println("Error while closing: " + e);
            }
        }
    }
}
```

**Problems with This Approach:**

- **More code** (need for `finally` block).
- **Possible memory/resource leak** if `br.close()` is not explicitly called.
- **Error handling complexity** (nested `try-catch` in `finally` block).

---

## Multiple Resources in `try-with-resources`

You can declare multiple resources in a single `try` block, separated by semicolons (`;`).

**Example**

```
import java.io.*;                                          Ummed Singh

public class MultipleResourcesExample {

    public static void main(String[] args) {

        try (

                FileReader fr = new FileReader("test.txt");

                BufferedReader br = new BufferedReader(fr)

        ) {

            System.out.println(br.readLine());

        } catch (IOException e) {
```

```
            System.out.println("Exception handled: " + e);

        }

    }

}
```

- Both `FileReader` and `BufferedReader` will be **automatically closed** after the `try` block.

---

### Custom Resources with `AutoCloseable`

You can create a custom resource that implements `AutoCloseable`.

**Example:**

```
class MyResource implements AutoCloseable {
    public void doSomething() {
        System.out.println("Resource is working");
    }

    @Override
    public void close() {
        System.out.println("Resource closed");
    }
}

public class CustomResourceExample {
    public static void main(String[] args) {
        try (MyResource res = new MyResource()) {
            res.doSomething();
        }
    }
}

Output:
Resource is working
Resource closed
```

- The `close()` method is called automatically when exiting the `try` block.

---

### Advantages of `try-with-resources`

✅ **Reduces Code Complexity** – No need for a `finally` block.
✅ **Prevents Resource Leaks** – Ensures resources are always closed.
✅ **Cleaner Exception Handling** – No need for nested `try-catch` blocks.
✅ **Supports Multiple Resources** – Can manage multiple resources in one statement.

---

## Creating Custom Exceptions

Custom exceptions allow developers to define application-specific errors by extending the `Exception` class. This approach improves error handling by providing meaningful exception messages and enabling structured exception management.

### Why Use Custom Exceptions?

- To represent **specific application errors**.
- To improve **code readability and debugging**.
- To distinguish between **different failure scenarios**.

### Example: Creating a Custom Exception

```
// Custom exception class                                    Ummed Singh
class CustomException extends Exception {
    public CustomException(String message) {
        super(message); // Pass message to Exception class
    }
}

public class CustomExceptionExample {
    // Method that throws the custom exception
    static void validate(int age) throws CustomException {
        if (age < 18) {
            throw new CustomException("Not eligible to vote");
        }
```

```
    }
    public static void main(String[] args) {
        try {
            validate(16); // This will throw CustomException
        } catch (CustomException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

## How It Works

1. **Define a Custom Exception**
   - Extend the `Exception` class (or `RuntimeException` for unchecked exceptions).
   - Call the `super()` constructor to pass the error message.
2. **Throw the Custom Exception**
   - The `validate()` method checks the age and throws `CustomException` if the condition is met.
3. **Handle the Exception**
   - The `try-catch` block catches and handles the exception.

## Checked vs. Unchecked Custom Exceptions

- **Checked Exception** (`extends Exception`): Must be handled using `try-catch` or declared using `throws`.
- **Unchecked Exception** (`extends RuntimeException`): Does not require mandatory handling.

### Example of Unchecked Custom Exception

```
class CustomUncheckedException extends RuntimeException {
    public CustomUncheckedException(String message) {
        super(message);
    }
}

public class UncheckedExceptionDemo {
    static void check(int number) {
        if (number < 0) {
```

```
            throw new CustomUncheckedException("Number must be
positive");
        }
    }

    public static void main(String[] args) {
        check(-5); // No need for try-catch, will terminate the
program
    }
}
```

# Testing Invariants Using Assertions

Assertions in Java are used to validate assumptions made in a program during development. They help catch logical errors early by **terminating execution** when an assumption is false. Assertions are primarily used for debugging and testing, rather than handling recoverable runtime errors.

---

## How Assertions Work

- The `assert` keyword is followed by a **boolean expression** that should evaluate to `true`.
- If the expression evaluates to `false`, an `AssertionError` is thrown.
- Assertions are **disabled by default** at runtime and need to be enabled explicitly using the `-ea` (enable assertions) JVM option.

---

## Example: Basic Assertion Usage

```
public class AssertionExample {                          Ummed Singh
    public static void main(String[] args) {
        int age = 15;
        assert age >= 18 : "Age must be at least 18";
        System.out.println("Age is " + age);
    }
}
```

**Explanation:**

- The assertion `assert age >= 18 : "Age must be at least 18";` checks whether `age` is at least 18.
- If `age < 18`, an `AssertionError` is thrown with the message `"Age must be at least 18"`.
- Otherwise, execution proceeds normally.

---

## Enabling Assertions in Java

Since assertions are disabled by default, running the above program normally will **not** trigger an assertion error. To enable assertions, run:

```
java -ea AssertionExample
```

(`-ea` stands for **enable assertions**)

If assertions are enabled and `age < 18`, the output will be:

```
Exception in thread "main" java.lang.AssertionError: Age must be at least 18
```

If assertions are **not enabled**, the program will execute normally:

```
Age is 15
```

---

## Using Assertions to Test Invariants

Assertions are useful for checking **invariants**—conditions that should always hold true throughout a program.

**Example: Validating a Bank Account Balance**

```
public class BankAccount {                                    Ummed Singh
    private int balance;

    public BankAccount(int balance) {
        assert balance >= 0 : "Balance cannot be negative";
```

```
        this.balance = balance;
    }

    public void withdraw(int amount) {
        assert amount > 0 : "Withdrawal amount must be positive";
        assert balance >= amount : "Insufficient balance";
        balance -= amount;
    }

    public int getBalance() {
        return balance;
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount(100);
        account.withdraw(150); // This will trigger an
AssertionError
    }
}
```

**Key Assertions:**

1. **Ensuring non-negative balance**: `assert balance >= 0 : "Balance cannot be negative";`
2. **Checking valid withdrawal amount**: `assert amount > 0 : "Withdrawal amount must be positive";`
3. **Preventing overdraft**: `assert balance >= amount : "Insufficient balance";`

If assertions are enabled and the withdrawal exceeds the balance, an `AssertionError` is thrown.

---

## When to Use Assertions

✅ **For development and debugging**
✅ **To check program invariants and assumptions**
✅ **For internal checks that should never fail**

---

## When NOT to Use Assertions

❌ **For handling user input validation** – Use `if` conditions and exceptions instead.
❌ **For essential program logic** – Assertions can be disabled, so critical checks should use `if-else` and throw exceptions.

**Incorrect Use (Bad Practice)**

```
public class UserInputExample {                                    Ummed Singh

    public static void main(String[] args) {

        int age = Integer.parseInt(args[0]);

        assert age >= 0 : "Age cannot be negative"; // ❌ Wrong!
Use exception handling instead.

        System.out.println("User's age: " + age);

    }

}
```

- Assertions should **not** replace input validation since they might be disabled at runtime.

**Instead, use:**

```
if (age < 0) {
    throw new IllegalArgumentException("Age cannot be negative");
}
```

---