Module 4

Java's Dual Forces: Exception Handling and Multi-threading

Exception Handling:

- Fundamentals
- Exception types
- Uncaught exceptions
- Using try and catch block
- Multiple catch clauses
- Nested try statements,
- throw, throws, finally
- Java’s Built-in Exceptions and
- User- defined Exceptions.

Multi-threading:

- Java thread model
- Main thread
- Thread life cycle
- Creating a thread
- Creating multiple threads
- Thread priorities
- Synchronization
- Inter-thread communication
- Suspending, Resuming and Stopping threads,
- Using multithreading.

Exception Handling Fundamentals:

What is an exception?

- **An Exception is an unwanted event that occurs during the execution of a program that interrupts the normal execution flow of the program.**

- It occurs when something unexpected happens, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.

- **When an exception occurs program execution gets terminated**. In such cases we get a system generated error message.

- Therefore, these exceptions are to be handled.

- **The Exception Handling in Java is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the program or application can be maintained.**

Example:

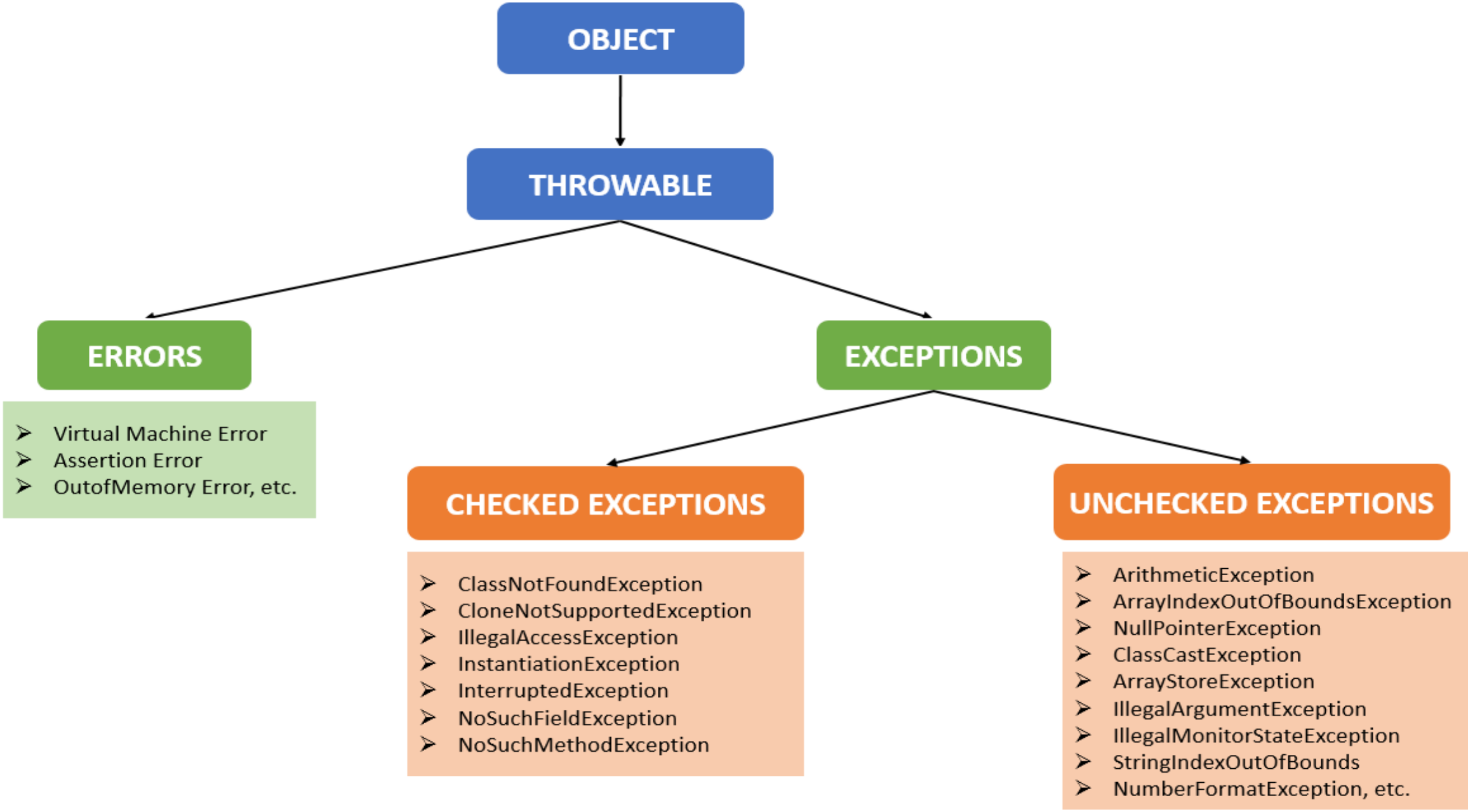
You are writing a program for division and both the numbers are entered by user. User can enter any number, if user enters the second number (divisor) as 0 then the **program will terminate and throw an exception because dividing a number by zero gives undefined result.**

Reasons for Exceptions in Java :

- A user has entered an invalid data.
- Trying to read/write a file that doesn’t exist or can’t be accessed.
- Division by zero.
- Trying to access or use a null object.
- Accessing elements beyond the valid range of an array.
- Trying to access restricted system resources.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.
- Resource Unavailability

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exception types:



Throwable Class

- The superclass of all errors and exceptions in Java.
- It has two direct subclasses:
  - **Error**
  - **Exception**

Errors Class:

This Class contains those errors which are difficult to handle. These indicate serious system problems, often JVM related, which an application might not be able to handle.
They occur during runtime of a program

Example:
StackOverflowError, OutOfMemoryError etc.

Exception class:

This class contains all the exceptions that can be handled easily.

There are two subclasses inherited it one is

1. **Runtime Exception(unchecked Exception)**
2. **checked Exception.**

Checked Exception

- Some exceptions are checked at the compile-time when the code is compiled. These are “Checked exceptions”. The Java program throws a compilation error when it finds that the code inside a program is error-prone.
- We can take care of the compilation errors thrown by checked exception by handling the exceptions by either enclosing the code in a try-catch block or using the throws keyword.

Examples: ClassNotFoundException, IOException, SQLException etc

Unchecked Exception:

- These types of exceptions occur during the runtime of the program. These are the exceptions that are not checked at a compiled time by the compiler.
- This Exception occurs due to bad programming.
- Can be handled with the help of try-catch Block.

Examples:

- Runtime Exceptions like IndexoutOfBoundException, Nullpointer Exception, etc

Uncaught Exceptions:

- When an exception occurs within a method, the method creates (throws) an object to describe the exception.
- This object is called as Exception Object, which contains the complete information about what went wrong here, ex: type of exception, state of program, line number in the source where this exception occurred and so on.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler :

1. Displays a string describing the exception
2. Prints a stack trace from the point at which the exception occurred, and
3. Terminates the program.

- **Java's default exception handling mechanism**

```
public class Test {
   public static void main(String[] args) {
   int a = 10;
   int b = 0;
   System.out.println("Before Division");
   double res = a/b;
   System.out.println("After Division");
   }
}
```

Output:
Before Division
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Test.main(Test.java:8)

Java Exception-Handling Mechanism:

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

try:

- The code that can cause the exception, is placed inside try block.
- The try block detects whether the exception occurs or not, if exception occurs, it transfer the flow of program to the corresponding catch block or finally block.
- A try block is always followed by either a catch block or finally block.

catch:

- The catch block is where we write the logic to handle the exception, if it occurs.
- A catch block only executes if an exception is caught by the try block. A catch block is always accompanied by a try block.

finally: This block always executes whether an exception occurs or not.

throw: It is used to explicitly throw an exception. It can be used to throw a checked or unchecked exception.

throws: It is used in method signature. It indicates that this method might throw one of the declared exceptions. While calling such methods, we need to handle the exceptions using try-catch block.

Using try and catch block:

Try catch block is used for exception handling in java.

The code (or set of statements) that can throw an exception is placed inside **try block** and if the exception is raised, it is handled by the corresponding **catch block**.

A try block is always followed by a catch block or finally block, if exception occurs, the rest of the statements in the try block are skipped and the control flow immediately jumps to the corresponding catch block.

Syntax:

```
try{
//statements that may cause an exception
}catch(Exception e){
//statements that will execute when exception occurs or statements to handle exception
}
```

Example: Handling Arithmetic Exceptions

```
class Main {
public static void main(String[] args) {
 try {
 // code that generate exception
 int divideByZero = 5 / 0;
System.out.println("Rest of code in try block");
 }
catch (ArithmeticException e) {
System.out.println("ArithmeticException => " + e.getMessage());
}
}
}
```

Output : ArithmeticException => / by zero

Explanation:

- In the example, we are trying to divide a number by 0. Here, this code generates an exception.
- To handle the exception, we have put the code, 5 / 0 inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped.
- The catch block catches the exception and statements inside the catch block is executed.
- If none of the statements in the try block generates an exception, the catch block is skipped.

Displaying a Description of an Exception

- **Throwable** overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception.
- You can display this description in a **println( )** statement by simply passing the exception as an argument.

- Eg:

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
}
```

Then each divide-by zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

Handling ArrayIndexOutOfBoundsException:

```
public class ArrayIndexExample {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30};
        try {
            // Trying to access 5th element (index 4), which doesn't exist
            System.out.println("Accessing element: " + numbers[4]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception Caught: " + e);
            System.out.println("Array index is out of bounds!");
        }
        System.out.println("Program continues after exception handling.");
    }
}
```

Output: Exception Caught:
java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 3
Array index is out of bounds!
Program continues after exception handling.

Handling NullPointerException

```
public class NullPointerExample {
    public static void main(String[] args) {
        String text = null;  // text is not initialized (null)

        try {
            // Attempting to call a method on a null object
            System.out.println("String length: " + text.length());
        } catch (NullPointerException e) {
            System.out.println("Exception Caught: " + e);
            System.out.println("You tried to access a method on a null object!");
        }

        System.out.println("Program continues after exception handling.");
    }
}
```

Java finally block

- **In Java, the finally block is always executed no matter whether there is an exception or not.**
- The finally block is optional. And, for each try block, there can be only one finally block.

Syntax:

```
try{
//statements that may cause an exception
}catch(Exception e){
 //statements that will execute if exception occurs
}finally{
//statements that execute whether the exception occurs or not
}
```

- ***If an exception occurs, the finally block is executed after the try...catch block. Otherwise, it is executed after the try block. For each try block, there can be only one finally block.***

Example Program:

```
class Main {
public static void main(String[] args) {
try {
 // code that generates exception
int divideByZero = 5 / 0;
 } catch (ArithmeticException e) {
System.out.println("ArithmeticException => " + e.getMessage());
 } finally {
System.out.println("This is the finally block always executed");
 }
 }
 }
```

Output

ArithmeticException => / by zero
This is the finally block always executed

Multiple catch clauses:

A catch block is where you handle the exceptions, this block must immediately placed after a try block.

- A single try block can have several catch blocks associated with it.

Syntax:

```
try {
    // Code that may throw multiple exceptions
} catch (ExceptionType1 e1) {
    // Handler for ExceptionType1
} catch (ExceptionType2 e2) {
    // Handler for ExceptionType2
}
// ... you can add more catch blocks
```

- To handle more than one exception which could be raised by a single piece of code, you can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.
- If no exception occurs in try block then the all catch blocks are completely ignored.

Note:

Generic catch block

```
catch(Exception e){
 //This catch block catches all the exceptions
}
```

A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them. In generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred.

Example Program:

```
public class MultiCatchExample {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
             Scanner s=new Scanner(System.in);
             int a=s.nextInt();
             int b=10/a; // May Throw Divide by Zero Exception
            arr[5] = 100;  // Throws ArrayIndexOutofBoundsException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception: " + e.getMessage());
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Out of Bounds: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("General Exception: " + e.getMessage());
        }
    }
}
```

Case 1: Input a= 0

•int b = 10 / a causes division by zero.

•Triggers **ArithmeticException**

Output:

Arithmetic Exception: / by zero

Case 2: Input a = 2

•b = 10 / 2 = 5 → no exception

•arr[5] = 100 → array of size 3 → index 5 out of bounds

Output:

Array Index Out of Bounds: Index 5 out of bounds for length 3

Key Rules:

1.Order matters – Catch more specific exceptions first, then the general ones like Exception.

Otherwise, you'll get a compile-time error.

2.You can’t catch the parent class (Exception) before the child class (ArithmeticException, etc.).

Example:

catch (Exception e) { ... }

catch (ArithmeticException e) { ... }

This is wrong: // Unreachable code

3.All catch blocks must have unique exception types.

A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.

```
/* This program contains an error.*/
class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
} catch(Exception e) {
System.out.println("Generic Exception catch.");
}

/* This catch is never reached because ArithmeticException is a subclass of Exception. */

catch(ArithmeticException e) { // ERROR - unreachable
System.out.println("This is never reached.");
}
}
}
```

Explanation:
If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught.

Nested try statements:

When a try catch block is present in another try block then it is called the nested try catch block.

```
....
//Main try block
try {
    statement 1;
    statement 2;
    //try-catch block inside another try block
    try {
        statement 3;
        statement 4;
        //try-catch block inside nested try block
        try {
            statement 5;
            statement 6;
        }
        catch(Exception e2) {
            //Exception Message
        }
    }
    catch(Exception e1) {
         //Exception Message
    }

}
//Catch of Main(parent) try block
catch(Exception e3) {
        //Exception Message
}
```

- Nested try blocks are useful for handling exceptions at different levels of code.
- If an exception occurs in a parent try block, the control jumps directly to the matching catch block in the parent or outer try block, and any nested try blocks are skipped.
- If an exception occurs in an inner try block and is not caught, it propagates to the outer try block.
- Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception and the system generated message would be shown for the exception.

```
class NestedExample {
    public static void main(String[] args) {
        try {// Outer try block
                System.out.println("Outer try block");
            try {// Inner try block 1
                System.out.println("Inner try block 1");
                int a = 10 / 0; // This will cause ArithmeticException
            } catch (ArithmeticException e) {
                System.out.println("Caught ArithmeticException in inner block 1");
            }
            try {// Inner try block 2
                System.out.println("Inner try block 2");
                int[] arr = new int[2];
                arr[5] = 100; // This will cause ArrayIndexOutOfBoundsException
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Caught ArrayIndexOutOfBoundsException in inner block 2");
            }
            System.out.println("End of outer try block");
        } catch (Exception e) {
            System.out.println("Caught Exception in outer block");
        }
        System.out.println("Program continues...");
    }
}
```

Output:

```
Outer try block
Inner try block 1
Caught ArithmeticException in inner block 1
Inner try block 2
Caught ArrayIndexOutOfBoundsException in inner bloc
End of outer try block
Program continues...
```

Example Program 2: Demonstrating Uncaught Exception in Inner Block

If an exception occurs in an inner try block and is not caught, it propagates to the outer try block.

```
class InnerBlockSkip {
    public static void main(String[] args) {
        try {
            System.out.println("Outer try block");
            try {
                System.out.println("Inner try block");
                int a = 10 / 0; // Causes ArithmeticException
            } catch (ArrayIndexOutOfBoundsException e) {
                // This catch block does NOT match ArithmeticException
                System.out.println("Caught ArrayIndexOutOfBoundsException in inner block");
            }
            System.out.println("This won't execute due to exception above");
        } catch (ArithmeticException e) {
            // Outer catch block catches the exception
            System.out.println("Caught ArithmeticException in outer block");
        }
        System.out.println("Program continues...");
    }
}
```

Output:

```
Outer try block
Inner try block
Caught ArithmeticException in outer block
Program continues...
```

throw:

- It is possible for a program to throw an exception explicitly, using the **throw** statement.
- The general form of **throw** is shown here:

throw *ThrowableInstance*;

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

Example:

throw new NullPointerException("This is a manually created exception");

- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception.
- If it finds a match, control is transferred to that statement.
- If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Example Program:

•Read two integers: numerator and denominator.

•If the denominator is zero, use throw to raise an ArithmeticException with the message:

"Error: Division by zero is not possible".Otherwise, perform the division and print the result as a double.

Program:

```
import java.util.Scanner;
public class DivisionCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int numerator = scanner.nextInt();
        int denominator = scanner.nextInt();
        try {
                if (denominator == 0) {
                throw new ArithmeticException("Error: Division by zero is not possible");
            }
                 else {
                double result = numerator / denominator;
                System.out.println(result);
            }
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Problem Statement:

Aanya is writing a simple banking program. She wants to ensure that users cannot enter a **negative withdrawal amount**. Write a Java program that:

•Reads an integer representing the withdrawal amount.

•If the amount is **negative**, explicitly **throw** an **IllegalArgumentException** using the throw keyword with the message:

"Withdrawal amount cannot be negative."

•If the amount is valid, print a success message.

```
import java.util.Scanner;
public class BankWithdrawal {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int amount = scanner.nextInt();
        try {
            // Check and throw exception if amount is negative
            if (amount < 0) {
                throw new IllegalArgumentException("Withdrawal amount cannot be negative.");
            }
                else
                {
                System.out.println("You have withdrawn: " + amount + " successfully.");
                }
        } catch (IllegalArgumentException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
          System.out.println("Transaction complete.");
    }
}
```

Explanation:

Inside the try block:

•It checks if the entered amount is **less than 0**.

•If **true**, it throws an IllegalArgumentException using the throw keyword with a custom error message.

If the amount is valid (0 or more), it prints a success message.

The catch block catches the thrown exception and prints the error message using getMessage().

throws:

- The throws keyword is used in a method declaration to indicate that the method might throw one or more exceptions that it does not handle itself.
- It tells the caller of the method to be ready to handle those exceptions.
- A throws clause lists the types of exceptions that a method might throw.
- All other exceptions that a method can throw must be declared in the throws clause
- **Syntax:**

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

exception-list is a comma-separated list of the exceptions that a method can throw.

Example Program:

You are writing a program that checks whether a person is eligible to vote. A person must be **18 years or older** to vote.

Write a Java program that:

- Reads an integer input for age.
- Defines a method called checkEligibility(int age) that:
  - **throws** an IllegalArgumentException if the age is less than 18.
  - Otherwise, prints "Eligible to vote".
- The main method should call this method and handle the exception using a try-catch block.
- If an exception occurs, print: "Exception caught: Age must be 18 or above."

```
import java.util.Scanner;
public class VotingCheck {
    // Method to check eligibility using throws
    static void checkEligibility(int age) throws IllegalArgumentException {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or above.");
        } else {
            System.out.println("Eligible to vote");
        }
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int age = scanner.nextInt();
        try {
            checkEligibility(age);  // Method call that might throw an exception
        } catch (IllegalArgumentException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
        System.out.println("Check complete.");
    }
}
```

```
import java.io.IOException;
public class ThrowVsThrows {
    // throws is used to declare the exception
    static void readFile(String fileName) throws IOException {
        if (fileName == null) {
            // throw is used to actually throw the exception
            throw new IOException("File name cannot be null.");
        } else {
            System.out.println("Reading file: " + fileName);
        }
    }
    public static void main(String[] args) {
        try {
            readFile(null);
        } catch (IOException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Explanation:

- The throws IOException part in the method signature **declares** that this method might throw an IOException.
- This is necessary for **checked exceptions** like IOException, so the compiler knows that whoever calls this method must handle the exception. i.e. here main() method must handle the exception using try-catch block

Java's Built-in Exceptions:

- Inside the standard package **java.lang**, Java defines several exception classes. Java has many built-in exceptions that are ready to use. These exceptions cover common error scenarios.
- The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.
- These exceptions need not be included in any method's **throws** list.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Checked Exceptions in java:

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

User Defined Exceptions or Custom Exceptions in Java:

Sometimes, predefined exceptions in Java are not suitable for describing a certain situation.

Imagine you're developing a **banking system**. If a user tries to **withdraw more money than the available balance**, using a predefined exception like IllegalArgumentException or ArithmeticException would not explain the real issue clearly.

In such cases, the programmer wants to create his own customized exception as per requirements of the application, which is called user-defined exception or custom exception in Java.

By creating a **custom exception**, you make the error message **more meaningful like** InsufficientFundsException.

User-defined exceptions in Java are those exceptions that are created by a programmer (or user) to meet the specific requirements of the application.

Examples:

1. A banking application, a customer whose age is lower than 18 years, the program throws a custom exception indicating "**needs to open a joint account**".

- 2. Voting age in India: If a person's age entered is less than 18 years, the program throws "**invalid age**" as a custom exception.

How to Create Your Own User-defined Exception in Java?

Step 1: User-defined exceptions can be created simply by extending the Exception class. This is done as:

```
class UserDefinedException extends Exception
```

Step 2: Provide constructors to initialize the exception with custom messages. Define one constructor that accepts a **String message**, so when the exception is thrown, a meaningful message is shown.

We call the **parent class constructor** using super(message) — this stores the message internally in the Exception class.

The message can be accessed later using getMessage() or our own method

```
UserDefinedException(String str)
{
  super(str); // Call superclass exception constructor and store variable "str" in it.
}
```

Step 3: In the last step, we need to create an object of the user-defined exception class and throw it using throw clause.

```
UserDefinedException obj = new UserDefinedException("Exception details");
  throw obj;
```

(or)

```
 throw new UserDefinedException("Exception details");
```

```
package customExceptionProgram;
public class InvalidAgeException extends Exception
{
// Declare a parameterized exception with string str as a parameter.
   InvalidAgeException(String str)
   {
      super(str);
   }
}
import java.util.Scanner;
public class TestClass
{
   private static int age;
   static void validate() throws InvalidAgeException
   {
   // Creating an object of Scanner class.
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter your age");
      age = sc.nextInt();
 if(age < 18)
         throw new InvalidAgeException("Invalid Age, You are not eligible to vote");
      else
         System.out.println("Welcome to vote");
    }
public static void main(String[] args)
{
   try
   {
      validate();
   }
   catch(Exception e)
   {
      System.out.println("Caught an Exception: \n "+e);
   }
 }
}
```

Output:

```
Enter your age 7
Caught an Exception:
customExceptionProgram.InvalidAgeException: Invalid Age, You are not eligible to vote
```

Program Statement:

Write a Java program to demonstrate the use of user-defined exceptions by implementing a bank account system.

The program should:

•Create a custom checked exception named InsufficientFundsException.

•Define a BankAccount class that maintains an account balance.

•Provide a withdraw(double amount) method that throws the custom exception if the withdrawal amount exceeds the current balance.

•In the main() method, attempt a withdrawal of an amount greater than the balance and handle the exception gracefully by displaying an appropriate message.

```
// Custom exception
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
// Bank class
class BankAccount {
    private double balance;
    public BankAccount(double balance) {
        this.balance = balance;
    }
    public void withdraw(double amount) throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException("Cannot withdraw ₹" + amount +
                ". Available balance: ₹" + balance);
        }
        balance -= amount;
        System.out.println("Withdrawal successful! Remaining balance: ₹" + balance);
    }
}
```

```
// Main class
public class BankApp {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(3000);

        try {
            account.withdraw(4000); // Trying to withdraw more
        } catch (InsufficientFundsException e) {
            System.out.println("Transaction Failed: " + e.getMessage());
        }
    }
}
```

Create a custom exception called **NegativeNumberException**. Write a method called **validateNumber** that takes an integer as input and throws **NegativeNumberException** if the number is negative.

Input format:

An integer representing a number.

Output format:

If the input number is positive, print the number to the console as "Number is valid: <number>".

- If the input number is negative, throw a **NegativeNumberException**.as "Error: Number cannot be negative"

```
class NegativeNumberException extends Exception {
    public NegativeNumberException(String message) {
        super(message);
    }
}
public class NumberValidator {
public static void validateNumber(int number) throws NegativeNumberException {
        if (number < 0) {
            throw new NegativeNumberException("Error: Number cannot be negative");
        } else {
            System.out.println("Number is valid: " + number);
        }
    }
public static void main(String[] args) {
        int input = -5;
        try {
            validateNumber(input);
        } catch (NegativeNumberException e) {
            System.out.println(e.getMessage());
        }
    }
}
```