CHAPTER NO:04
Exception Handling {10 MARKS}

4.1 Exception-Handling Fundamentals, Exception class hierarchy

4.2 Uncaught Exceptions, try and catch, Multiple catch Clauses, nested try Statements, throw, throws, finally clauses

4.3 Java's Built-in Exceptions - Checked and Unchecked Exceptions

4.4 Creating Your Own Exception Subclasses using Exceptions

4.1 Exception-Handling Fundamentals, Exception class hierarchy

What is an Exception in Java?

- An **exception** is an object that represents an error or unexpected event that occurs during program execution.
- When an exception occurs, normal flow of the program is disrupted.

Why Handle Exceptions?

- To **prevent the program from crashing** unexpectedly.
- To **handle errors gracefully** and provide meaningful responses.
- To ensure **resources** (files, connections) are properly managed and closed.

Java Exception Handling Constructs:

- **try**: Block of code that might throw an exception.
- **catch**: Block to handle the exception if thrown.
- **finally**: Block that always executes after try/catch, often used for cleanup.
- **throw**: Used to explicitly throw an exception.
- **throws**: Used in method signature to declare that the method may throw exceptions.

Example (in Java):

java

```java
try {
    int result = 10 / 0;  // This causes ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero.");
} finally {
    System.out.println("Execution completed.");
}
```

```
try {
    int result = 10 / 0;  // This causes ArithmeticException
```

- This line tries to divide `10` by `0`, which is **not allowed** in Java.
- It causes an `ArithmeticException` to be thrown at runtime.
- Since the division fails, the next lines in the `try` block are skipped, and control goes to the `catch` block.

```
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero.");
```

- This block **catches** the `ArithmeticException` that was thrown in the `try` block.
- The message `"Cannot divide by zero."` is printed to inform the user about the error.
- The variable `e` holds the exception object (you can use it to get the message or stack trace if needed).

```
} finally {
    System.out.println("Execution completed.");
}
```

- The `finally` block **always executes**, regardless of whether an exception occurred or not.
- Here, it prints `"Execution completed."`, ensuring that any necessary cleanup or final steps are always performed.

🖨️ Output of the Program

When you run this code, the output will be:

```csharp

Cannot divide by zero.
Execution completed.
```

Exception Class Hierarchy:

- Exceptions are organized in a class hierarchy, typically something like this (using Java terminology).
- This picture shows the **Java Exception Hierarchy** — how errors and exceptions are organized in Java.

```
java.lang.Object
    └── java.lang.Throwable
        ├── java.lang.Error                    (Serious errors, JVM related)
        └── java.lang.Exception
            ├── java.lang.RuntimeException  (Unchecked exceptions)
            └── Checked Exceptions           (Must be handled or declared)
```

Step-by-step explanation:

1. **Throwable**
   - This is the top class for all errors and exceptions in Java.
   - Anything that can be "thrown" in Java (like an error or exception) is a subclass of `Throwable`.
2. **Error**
   - Serious problems related to the Java Virtual Machine (JVM) or system.
   - Examples: Out of memory, JVM crash.
   - These are usually **not handled** by programs because they are serious system problems.
3. **Exception**
   - Problems that happen during program execution that the program **can** handle.
   - These are things like trying to open a file that doesn't exist, or divide by zero.

4. Inside **Exception**, there are two types:
    - **RuntimeException (Unchecked exceptions):**
        - These happen during program execution due to bugs or wrong input.
        - You **don't have to** handle these explicitly in your code, but you can if you want to.
        - Examples: Null pointer exception, array index out of bounds.
    - **Checked exceptions:**
        - These are exceptions Java **forces you to handle** in your code by either catching them or declaring them in the method signature.
        - Examples: File not found, input/output errors.

4.3 Java’s Built-in Exceptions - Checked and Unchecked Exceptions:

✅ 4.3 Java’s Built-in Exceptions

Java provides a rich set of **built-in exception classes** in the `java.lang` package, which help in handling common runtime errors.

These exceptions are categorized into two main types:

📘 1. Checked Exceptions (Compile-Time Exceptions)

✔️ Definition:

- Exceptions that **must be either caught or declared** in the method using `throws`.
- Checked by the compiler during **compile time**.
- Usually represent **external problems** that your code can recover from.

🔍 Common Checked Exceptions:

Exception	Description
`IOException`	Input/output operation failure (e.g., file not found).
`SQLException`	Database access error.
`ClassNotFoundException`	Requested class not found at runtime.
`FileNotFoundException`	File cannot be found during read/write operation.
`InterruptedException`	Thread interruption during sleep or wait.

```java
import java.io.*;

public class CheckedExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("data.txt"); // May throw FileNotFoundException
        } catch (FileNotFoundException e) {
            System.out.println("File not found.");
        }
    }
}
```

📕 2. Unchecked Exceptions (Runtime Exceptions)

✔️ Definition:

- Exceptions that are **not checked at compile time**.
- Usually caused by **programming mistakes** like logic errors or improper use of APIs.
- Subclasses of `RuntimeException`.

🔍 Common Unchecked Exceptions:

Exception	Description
`ArithmeticException`	Arithmetic error like divide by zero.
`NullPointerException`	Accessing method or variable from a null object.
`ArrayIndexOutOfBoundsException`	Accessing invalid array index.
`IllegalArgumentException`	Invalid method argument passed.
`NumberFormatException`	Improper conversion from string to number.

```java
public class UncheckedExample {
    public static void main(String[] args) {
        int a = 10 / 0;  // ArithmeticException: divide by zero
        System.out.println("This line won't execute.");
    }
}
```

4.2 Uncaught Exceptions, try and catch, Multiple catch Clauses, nested try Statements, throw, throws, finally clauses

Uncaught Exceptions:

- When an exception happens but you do not handle it using try-catch, it is called an uncaught exception.
- This usually causes the program to stop (crash) and show an error message.
- **Example:** If you divide by zero without handling it, your program crashes.

2. try and catch

- To handle exceptions, you use a try block to put the code that might cause an exception.
- Right after try, you add one or more catch blocks to handle specific exceptions if they happen.
  - Syntax:

```java
try {
    // code that might throw an exception
} catch (ExceptionType e) {
    // code to handle the exception
}
```

3. Multiple catch Clauses

- You can catch different types of exceptions separately by writing multiple catch blocks after a single try.
- This helps to handle each exception in its own way.

Example:

java

```java
try {
    // code that may throw IOException or ArithmeticException
} catch (IOException e) {
    // handle IO exception
} catch (ArithmeticException e) {
    // handle arithmetic exception
}
```

4. Nested try Statements

- You can put a try-catch block inside another try block.
- This is called nested try.
- It allows you to handle different exceptions at different levels

Example:

```java
try {
    // outer try
    try {
        // inner try
    } catch (Exception e) {
        // handle inner exception
    }
} catch (Exception e) {
    // handle outer exception
}
```

5. Throw

- The throw keyword is used to explicitly throw an exception from your code.
- You create an exception object and then throw it.

Example:

```java
throw new ArithmeticException("Divide by zero error");
```

6. Throws

The throws keyword is used in a method declaration to indicate that this method might throw certain exceptions.

It tells the caller of the method that they must handle or declare these exceptions.

Example:

```java
public void readFile() throws IOException {
    // code that might throw IOException
}
```

7. finally clause

The finally block is optional but very useful.

It contains code that will always run after the try and catch blocks, whether an exception was thrown or not.

It’s often used for cleanup, like closing files or releasing resources.

Example:

java

```java
try {
    // code that might throw exception
} catch (Exception e) {
    // handle exception
} finally {
    // always executes
    System.out.println("Cleaning up...");
}
```

Uncaught Exceptions	Exceptions that are not handled; crash program
try	Block of code where exception might occur
catch	Block to handle the exception
Multiple catch	Handling different exceptions separately
Nested try	try inside another try for layered handling

🔶 1. `throw` — *Used to actually throw an exception*

- You use `throw` when you want to **manually create and throw** an exception.
- It is used **inside** a method or block.

✅ Syntax:

```java
throw new ExceptionType("message");
```

✅ Example:

```java
public void checkAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Underage not allowed");
    }
}
```

🔶 2. `throws` — *Used to declare an exception*

- You use `throws` in the **method declaration** to say:

> "This method might throw an exception, so whoever calls it must handle it."

✅ Syntax:

```java
public void readFile() throws IOException {
    // code that may throw IOException
}
```

✅ Example:

```java
public void divide(int a, int b) throws ArithmeticException {
    if (b == 0) {
        throw new ArithmeticException("Cannot divide by zero");
    }
    System.out.println(a / b);
}
```

🔶 3. `finally` — *Used to write code that always runs*

- You use `finally` with `try-catch` to write code that should **always run** — whether an exception happens or not.
- Usually used to **clean up resources**, like closing files or network connections.

✅ Syntax:

java Copy code

```java
try {
    // code that may throw exception
} catch (Exception e) {
    // handle exception
} finally {
    // always runs
}
```

✅ **Example:**

java

Copy code

```java
try {
    int result = 10 / 2;
} catch (Exception e) {
    System.out.println("Error occurred");
} finally {
    System.out.println("This always runs!");
}
```

➡️ **Key point**: `finally` block **always executes**, even if an exception is not thrown or caught.

Keyword	Purpose	Where it's used	Example
`throw`	Actually throws an exception	Inside a method	`throw new Exception();`
`throws`	Declares that method can throw	In method declaration	`void m() throws IOException`
`finally`	Always executes cleanup code	After try-catch block	In `try-catch-finally` block

```
public class FinallyExample {

    public static void main(String[] args) {
        System.out.println("Program started");

        int a = 10;
        int b = 0; // This will cause an exception

        try {
            int result = a / b; // Division by zero!
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println(" Exception caught: " + e.getMessage());
        } finally {
            // This block ALWAYS runs
            System.out.println("finally block: Cleaning up resources...");
        }

        System.out.println("Program ended");
    }
}
```