DAY- 6

Exceptions in Java

1. What are Exceptions in Java?

An **exception** is an event that disrupts the normal flow of a program's execution. In Java, exceptions are objects that represent errors or unusual conditions that occur during program execution. They provide a mechanism to handle errors gracefully without terminating the program abruptly.

2. Types of Exceptions

Java exceptions are categorized into two primary types:

- **Checked Exceptions**
- **Unchecked Exceptions**

3. Checked Exceptions

- **Definition**: Checked exceptions are exceptions that the compiler forces the programmer to handle. These exceptions are checked at **compile-time**.
- **Key Point**: The Java compiler requires that checked exceptions be either:
  1. **Caught** using a `try-catch` block.
  2. **Declared** in the method signature using the `throws` keyword if the method can throw this type of exception.
- **Examples**:
  - `IOException`: Happens when there is an I/O operation failure (e.g., file not found).
  - `SQLException`: Occurs when interacting with a database.
  - `ClassNotFoundException`: Happens when an application tries to load a class but cannot find it.
- **Why Use Checked Exceptions**: They are used to handle conditions that a program might want to recover from, such as a file being unavailable or a network connection failing. Since these conditions are expected, it is mandatory to plan for them.
- **Handling Checked Exceptions**:
  1. **Using try-catch**:

```
try {
    // Code that may throw a checked exception
    FileReader fr = new FileReader("file.txt");
} catch (IOException e) {
    // Handling the exception
    System.out.println("File not found!");
}
```

  2. **Using throws**:

```
public void readFile() throws IOException {
    FileReader fr = new FileReader("file.txt");
}
```

4. Unchecked Exceptions

- **Definition**: Unchecked exceptions are exceptions that the compiler does not require you to handle. These exceptions are checked **at runtime**.
- **Key Point**: Unchecked exceptions do not need to be declared in the method signature nor do they need to be caught explicitly. They are typically the result of programming errors.
- **Examples**:

- `NullPointerException`: Occurs when you try to use a `null` reference where an object is expected.
- `ArrayIndexOutOfBoundsException`: Happens when an invalid index is accessed in an array.
- `ArithmeticException`: Occurs when there is a mathematical error (e.g., dividing by zero).

- **Why Use Unchecked Exceptions**: Unchecked exceptions are used for errors that usually cannot be anticipated, such as logic errors or invalid operations that should be fixed in the code rather than handled during runtime.
- **Handling Unchecked Exceptions**:
  - It's optional to catch these exceptions, but in some cases, you may want to catch and handle them to prevent your program from crashing.

```
public class Test {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // Throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
    }
}
```

5. Difference Between Checked and Unchecked Exceptions

- **Checked Exceptions**:
  - Are checked at **compile-time**.
  - Must be **caught** or **declared**.
  - Represent conditions that the program can recover from (e.g., file not found, I/O errors).
- **Unchecked Exceptions**:
  - Are checked at **runtime**.
  - Do **not need** to be caught or declared.
  - Represent programming errors (e.g., null pointer access, array index out of bounds).

Type of Exception	Checked Exceptions	Unchecked Exceptions
Checked at	**Compile-time**	**Runtime**
Must be handled	**Yes**, must be caught or declared	**No**, optional to handle
Typical Cause	Conditions that can be recovered from (e.g., file not found, I/O errors)	Programming errors (e.g., null pointer, array index out of bounds)
Example	`IOException`, `SQLException`	`NullPointerException`, `ArrayIndexOutOfBoundsException`

6. How to Handle Exceptions

- **Try-Catch Block**: Used to handle exceptions by attempting code execution in the `try` block and catching exceptions in the `catch` block.

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handle the exception
}
```

- **Finally Block**: Executes code after the `try` and `catch` blocks, regardless of whether an exception occurred or not.

```
try {
    // Code
} catch (Exception e) {
    // Handle exception
} finally {
    // Code that always runs
}
```

- **Throws Keyword**: Used to declare exceptions that a method may throw.

```
public void someMethod() throws IOException {
    // Code that may throw IOException
}
```

7. Custom Exceptions

- **Creating Custom Exceptions**: You can create your own exception classes by extending the `Exception` or `RuntimeException` class.

```
public class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}
```

Summary

- **Checked Exceptions** are those that the compiler forces you to handle. They represent situations that can be anticipated and recovered from (e.g., file I/O, database errors).
- **Unchecked Exceptions** are not checked by the compiler, usually represent bugs, and typically indicate serious errors in the program that should be fixed.
- Exception handling in Java is crucial for writing robust programs that can handle unexpected situations without crashing.

Difference Between `throws` and `catch`

1. `throws`:
   - **Declares** that a method **might throw** an exception.
   - **Passes the responsibility** of handling the exception to the calling method or the JVM (for unchecked exceptions).
   - It **does not catch or handle** the exception. It simply informs the compiler that this method may throw an exception.
   - Used mainly for **checked exceptions** that you want to propagate further up the call stack.
2. `catch`:
   - **Handles** the exception that is thrown within a `try` block.
   - It **captures** the exception and allows you to **define how to respond** to it, such as printing an error message, logging the error, or taking corrective action.
   - `catch` is used to **recover from** exceptions and keep the program running, instead of letting it crash.

Example: Using `throws` vs `catch`

Using `throws` (Declaration, No Handling in Method)

```
import java.io.FileNotFoundException;
import java.io.File;
import java.util.Scanner;

public class ThrowsExample {

    // Declaring that this method might throw a FileNotFoundException
    public static void readFile(String fileName) throws FileNotFoundExcepti
on {
        File file = new File(fileName);
        Scanner scanner = new Scanner(file);  // Might throw FileNotFoundEx
ception
        while (scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }
        scanner.close();
    }

    public static void main(String[] args) {
        try {
            // Calling readFile, which might throw an exception
            readFile("nonexistentfile.txt");  // Delegates exception handli
ng to main
        } catch (FileNotFoundException e) {
```

```
            System.out.println("Exception caught in main: File not foun
d!");
        }
    }
}
```

Explanation:

- `readFile` declares `throws FileNotFoundException`. This means `readFile` doesn't handle the exception itself, but the caller (in this case, `main`) is responsible for handling it.
- In `main`, we use a `try-catch` block to handle the exception.

Using `catch` (Handling the Exception Inside the Method)

```
import java.io.FileNotFoundException;
import java.io.File;
import java.util.Scanner;

public class CatchExample {

    // Handling the exception inside the method using try-catch
    public static void readFile(String fileName) {
        try {
            File file = new File(fileName);
            Scanner scanner = new Scanner(file);  // Might throw FileNotFou
ndException
            while (scanner.hasNextLine()) {
                System.out.println(scanner.nextLine());
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("Exception caught inside method: File not fo
und!");
        }
    }

    public static void main(String[] args) {
        readFile("nonexistentfile.txt");  // Handles exception internally w
ithin the method
    }
}
```

Explanation:

- `readFile` handles the exception internally using a `try-catch` block.
- The `main` method doesn't need to declare `throws` or handle the exception because it's already handled in `readFile`.

Summary:

- `throws` simply **declares** that a method may throw an exception, and **delegates** the responsibility of handling it to the calling method.
- `catch` is used inside a `try-catch` **block** to **handle** the exception by taking corrective actions, ensuring the program can continue running smoothly.

THROW

In Java, the `throw` keyword is used to explicitly throw an exception. It is typically used inside a method or a block of code to indicate that an exceptional condition has occurred. When `throw` is used, it creates an instance of an exception and hands control to the Java runtime system to handle the exception (or propagate it further if not caught).

Syntax:

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

Here, `ExceptionType` is a class that inherits from `Throwable` (like `Exception` or `RuntimeException`), and `"Error message"` is the custom message that provides more details about the exception.

Example:

```
public class ThrowExample {
    public static void main(String[] args) {
        try {
            checkAge(15);  // This will throw an exception
        } catch (Exception e) {
            System.out.println(e.getMessage());  // Catch and print the err
or message
        }
    }

    static void checkAge(int age) throws Exception {
        if (age < 18) {
            throw new Exception("Age must be 18 or older.");
        }
    }
}
```

Key points about `throw`:

- It is used to throw a single exception manually.
- The exception can be a checked or unchecked exception.

- The method that throws a checked exception must either handle it using a `try-catch` block or declare it using `throws`.