# Exception Handling

Exception handling in Java allows developers to manage runtime errors, ensuring that the normal flow of a program is maintained even when errors occur. Without exception handling, errors would cause abrupt termination of the program, which is undesirable in most applications. Exception handling provides a robust mechanism for managing errors, allowing developers to deal with unexpected conditions effectively.

**What is an Exception?**

An **exception** is an event that occurs during the execution of a program and disrupts its normal flow. In Java, exceptions are objects that encapsulate error information. Exceptions occur due to several reasons:

- Invalid user input (e.g., entering a letter when a number is expected).
- File not found.
- Division by zero.
- Network errors.

**Exception Class Hierarchy**

Exceptions in Java are objects, and they are derived from the `java.lang.Throwable` class. The `Throwable` class has two main subclasses:

1. **Error**: Represents serious system-level errors that typically cannot be handled by the program (e.g., `OutOfMemoryError`, `StackOverflowError`). Errors indicate problems that the application should not try to catch.
2. **Exception**: Represents conditions that applications might want to catch and handle. `Exception` is further divided into:
    - **Checked Exceptions**: Exceptions that the compiler requires to be handled or declared using `throws` (e.g., `IOException`, `SQLException`).
    - **Unchecked Exceptions**: Also known as runtime exceptions. These are not checked at compile time, and occur due to programming errors like logic mistakes (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`).

**Exception Hierarchy in Java:**

```
java.lang.Throwable
    ├── java.lang.Error
    └── java.lang.Exception
        ├── java.lang.RuntimeException
        └── other exceptions (Checked Exceptions)
```

**Types of Exceptions**

1. **Checked Exceptions**: These are exceptions that a programmer must handle in the code. If a checked exception is not caught or declared in the method signature, the program will not compile. These are typically external issues, such as file handling errors or network problems.
    - **Examples**: `IOException`, `SQLException`, `FileNotFoundException`.

    **Example of Checked Exception:**

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

    public class CheckedExceptionExample {
        public static void main(String[] args) {
            try {
                File file = new File("nonexistentfile.txt");
                Scanner reader = new Scanner(file);  // May throw FileNotFoundException
            } catch (FileNotFoundException e) {
                System.out.println("File not found.");
            }
        }
    }
    ```
2. **Unchecked Exceptions (Runtime Exceptions)**: These exceptions occur at runtime and are usually due to logical errors or improper use of API. Unchecked exceptions are not checked at compile-time.

- **Examples**: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`.

**Example of Unchecked Exception:**

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] arr = new int[3];
        System.out.println(arr[5]);  // Will throw ArrayIndexOutOfBoundsException
    }
}
```

3. **Errors**: These are serious problems that applications should not try to catch (e.g., system-related errors). Errors are usually not recoverable, such as JVM crashes, running out of memory, etc.
   - **Examples**: `OutOfMemoryError`, `StackOverflowError`.

---

**Exception Handling Mechanism in Java**

Java provides several keywords for handling exceptions:

1. **try**: The code that might throw an exception is placed inside the `try` block.
2. **catch**: The code that handles the exception is placed inside the `catch` block.
3. **finally**: A block of code that always executes, whether an exception is thrown or not. It is typically used for resource cleanup (e.g., closing file or database connections).
4. **throw**: Used to explicitly throw an exception.
5. **throws**: Used to declare that a method might throw an exception. This is required for checked exceptions that the method does not handle.

---

**Basic Example of Exception Handling**

```
public class BasicExceptionHandling {
    public static void main(String[] args) {
        try {
            int division = 10 / 0;  // ArithmeticException: Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero.");
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

**Explanation**:

- **try block**: The risky code (division by zero) is placed here.
- **catch block**: Catches the `ArithmeticException` and handles it by displaying a message.
- **finally block**: Executes regardless of whether an exception is thrown or not, ensuring that any cleanup code runs.

**Multiple Catch Blocks**

Java allows multiple `catch` blocks to handle different exceptions separately.

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length());  // This will throw NullPointerException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception caught");
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught");
        } catch (Exception e) {
            System.out.println("General Exception caught");
        }
    }
}
```

**Explanation**:

- Multiple `catch` blocks handle specific exceptions separately.
- The general `Exception` block will catch any exception not handled by earlier blocks.

**The `finally` Block**

The `finally` block always executes, regardless of whether an exception is thrown or not. It is often used to close resources like files, network connections, or databases.

```
public class FinallyBlockExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 2;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error occurred.");
        } finally {
            System.out.println("Finally block always executes.");
        }
    }
}
```

**Explanation**:

- In this example, the `finally` block will execute even though no exception occurs.

**Throwing Exceptions Using `throw`**

The `throw` keyword is used to explicitly throw an exception. This is useful for throwing custom exceptions.

```
public class ThrowExample {
    public static void main(String[] args) {
        checkEligibility(15);
    }

    static void checkEligibility(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        } else {
            System.out.println("Eligible to vote");
        }
    }
}
```

**Explanation**:

- The `checkEligibility` method throws an `ArithmeticException` if the age is less than 18.

**Using `throws` to Declare Exceptions**

The `throws` keyword is used to declare that a method might throw an exception but does not handle it.

```
import java.io.IOException;

public class ThrowsExample {
    public static void main(String[] args) throws IOException {
        validateFile();
    }

    static void validateFile() throws IOException {
        throw new IOException("File not found");
    }
}
```

**Explanation**:

- The `validateFile` method throws an `IOException` which is not handled but declared using the `throws` keyword.

**Custom Exceptions**

Java allows developers to create their own exceptions by extending the `Exception` class.

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

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }

    static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age less than 18 is not allowed.");
        }
    }
}
```

**Explanation**:

- `InvalidAgeException` is a custom exception that is thrown if the age is less than 18.

**Exception Propagation**

When an exception is thrown, it is propagated up the call stack until it is caught by an appropriate `catch` block.

```
public class ExceptionPropagationExample {
    void method1() {
        int data = 10 / 0;  // ArithmeticException occurs here
    }

    void method2() {
        method1();  // Exception propagates to this method
    }

    public static void main(String[] args) {
        ExceptionPropagationExample obj = new ExceptionPropagationExample();
        try {
            obj.method2();
        } catch (ArithmeticException e) {
            System.out.println("Exception caught in main method");
        }
    }
}
```

**Explanation**:

- The exception occurs in `method1()` and is propagated to `method2()` and then to the `main()` method, where it is caught.

**Best Practices for Exception Handling**

1. **Use specific exceptions**: Always catch specific exceptions, rather than using a general `Exception` block.
2. **Clean up resources in finally**: Use the `finally` block to release resources, such as closing files, network connections, or database connections.
3. **Don't ignore exceptions**: Avoid using empty catch blocks; always handle exceptions meaningfully.
4. **Don't use exceptions for flow control**: Exceptions should be used for handling exceptional conditions, not for normal program flow.
5. **Throw custom exceptions**: Create meaningful custom exceptions that convey the cause of the error clearly.