Herat University
Computer Science Faculty
Software Engineering Department

Introduction to Exceptions and Basic Handling

Seyyed Kamran Hosseini

3/10/2025

Table Content

Introduction to Exceptions

- What are Exceptions?
  - Exceptions are events that disrupt the normal flow of a program.
  - They occur when something unexpected happens (e.g., dividing by zero, accessing a null object).
- Why Do Exceptions Occur?
  - Programming errors (e.g., logic errors, invalid input).
  - External factors (e.g., file not found, network issues).

Continue…

- Examples of Common Exceptions:
  - ArithmeticException: Division by zero.
  - NullPointerException: Accessing a null object.
  - ArrayIndexOutOfBoundsException: Accessing an invalid array index.
- Example:

```
int result = 10 / 0; // This will throw ArithmeticException
```

Types of Exceptions

- Checked Exceptions:
  - Checked at compile-time.
  - Must be handled using try-catch or declared with throws.
  - Example: IOException (file not found).
- Unchecked Exceptions:
  - Checked at runtime.
  - Optional to handle.
  - Example: NullPointerException, ArithmeticException.
- Example:

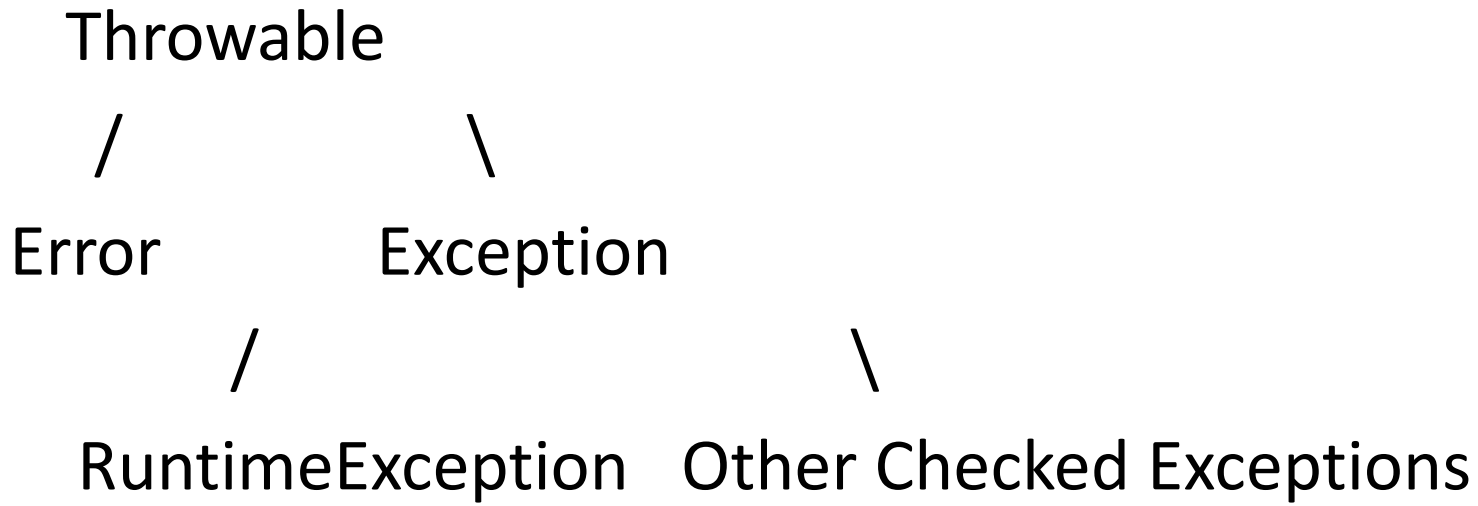
```
String str = null;
System.out.println(str.length()); // This will throw NullPointerException
```

Exception Hierarchy

- Java's exception handling is built upon a hierarchical structure that helps organize and classify different types of exceptions. Let's explore the key components of the exception hierarchy.

1. Throwable:
   - At the top of the exception hierarchy is the Throwable class.
   - Both Error and Exception classes extend from Throwable.
2. Exception:
   - The Exception class is the parent class for all checked exceptions.
   - It further divides into subclasses such as IOException, SQLException, etc.
3. RuntimeException:
   - RuntimeException is the parent class for all unchecked exceptions.
   - Exceptions like NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException extend from this class.

Java Exception Class Hierarchy



Handling Exceptions

- The try-catch Block
- Exception handling in Java revolves around the try-catch block, providing a structured way to manage potential errors during program execution.

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

Continue…

- The try block encloses the code that might throw an exception.
- The catch block(s) handle specific types of exceptions, providing a recovery mechanism.
- The finally block contains code that always executes, regardless of whether an exception occurs or not. It is optional.

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.err.println("Error: Division by zero");
        }
    }
    private static int divide(int numerator, int denominator) {
        return numerator / denominator;
    }
}
```

Multiple catch Blocks

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Handle ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle ExceptionType2
}
```

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0); // Potential division by zero
            System.out.println("Result: " + result); // This line won't be reached if an exception
occurs
        } catch (ArithmeticException e) {
            System.err.println("Error: Division by zero");
        } catch (Exception e) {
            System.err.println("An unexpected error occurred: " + e.getMessage());
        }
        // Code after the try-catch block
        System.out.println("Program continues after exception handling");
    }
    private static int divide(int numerator, int denominator) {
        return numerator / denominator; // Potential division by zero
    }
}
```

```
import java.util.Scanner;

public class UserInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter a number: ");
            int number = scanner.nextInt(); // Potential InputMismatchException
            System.out.println("You entered: " + number);
        } catch (java.util.InputMismatchException e) {
            System.err.println("Error: Invalid input. Please enter a valid number.");
        }
        System.out.println("Program continues after user input handling");
    }
}
```

The finally Block

- In Java, the finally block is a crucial component of the try-catch mechanism, providing a way to execute code that must run, regardless of whether an exception occurs or not.

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Handle the exception
} finally {
    // Code that always executes, whether an exception occurs or not
}
```

- The finally block contains code that is guaranteed to be executed.
- It is often used for cleanup operations, such as closing resources (files, database connections) or releasing locks.

```
import java.io.FileReader;
import java.io.IOException;

public class FinallyExample {
    public static void main(String[] args) {
        FileReader fileReader = null;

        try {
            fileReader = new FileReader("example.txt");
            // Code that reads from the file
        } catch (IOException e) {
            System.err.println("Error reading the file: " + e.getMessage());
        } finally {
            try {
                if (fileReader != null) {
                    fileReader.close(); // Ensure the FileReader is closed
                }
            } catch (IOException e) {
                System.err.println("Error closing the file: " + e.getMessage());
            }
        }
        System.out.println("Program continues after file handling");
    }
}
```

Examples of Exceptions

Example 1: Handling Division by Zero

```
public class Main {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will throw ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero!");
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

Example 2: Handling Invalid Array Access

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        try {
            System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Invalid array index!");
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

Time for Activity

Student Activity

- Activity 1: Division by Zero
  - Write a program that takes two numbers as input and handles division by zero.
  - Use try-catch to handle the exception.
- Activity 2: Invalid Array Access
  - Write a program that accesses an array and handles invalid indices.
  - Use try-catch to handle the exception.

Any Question?