Unit 5: Handling Error/Exception in Java

1. Basic Exceptions

In Java, **exceptions** are events that disrupt the normal flow of a program. They occur during the execution of a program when something unexpected happens, such as dividing by zero, accessing an invalid index in an array, or trying to open a file that doesn't exist. Java provides a robust mechanism to handle exceptions using **try-catch blocks**, **throw**, and **throws** keywords.

Types of Exceptions

1. **Checked Exceptions**:
    - These are exceptions that are checked at **compile-time**.
    - Examples: `IOException`, `SQLException`, `ClassNotFoundException`.
    - The programmer must handle these exceptions using `try-catch` blocks or declare them using the `throws` keyword.
2. **Unchecked Exceptions**:
    - These are exceptions that are checked at **runtime**.
    - Examples: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.
    - Unchecked exceptions are not required to be explicitly handled using `try-catch`, declared with `throws`, or manually thrown using `throw`. The Java runtime automatically detects and throws these exceptions when specific conditions occur, such as division by zero. However, it is considered good practice to use `try-catch` to handle them gracefully and prevent unexpected program crashes.
3. **Errors**:
    - These are severe issues that are not meant to be handled by the program.
    - Examples: `OutOfMemoryError`, `StackOverflowError`.

Exception Handling Mechanism

Java provides five keywords for exception handling:

1. `try`: A block of code where exceptions might occur.
2. `catch`: A block that handles the exception.
3. `finally`: A block that executes regardless of whether an exception occurs.
4. `throw`: Used to explicitly throw an exception.
5. `throws`: Used to declare exceptions that a method might throw.

Common Exceptions in Java

1. **ArithmeticException**: Occurs when dividing by zero.
2. **NullPointerException**: Occurs when trying to access a null object.
3. **ArrayIndexOutOfBoundsException**: Occurs when accessing an invalid array index.
4. **IOException**: Occurs during input/output operations.
5. **NumberFormatException**: Occurs when converting a string to a numeric type fails.

Lab 1: Basic Exception

```
public class Main {
    public static void main(String[] args) {
```

```
        int a = 10;
        int b = 0;
        try {
            int result = a / b; // This will cause an ArithmeticException and
automatically throws an ArithmeticException
            System.out.println("Result is " + result);// This line will not execute
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");// Exception is caught here -
> Handling exception here
        }
    }
}
```

Output:

```
Cannot divide by zero!
```

Explanation:

- The code attempts to divide `a` by `b`, which is zero, leading to an `ArithmeticException`.
- The `try` block contains the code that might throw an exception.
- The `catch` block catches the exception and handles it by printing an error message.

---

2. Proper Use of Exceptions

Exceptions should be used for exceptional conditions only, not for regular control flow. They are costly in terms of performance and should not be used to handle predictable conditions that can be checked with simple if-else statements.

Lab 2: Proper Use of Exceptions

```
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        int index = 3;
        if (index >= 0 && index < numbers.length) {
            System.out.println("Number at index " + index + " is " + numbers[index]);
        } else {
            System.out.println("Index out of bounds!");
        }
    }
}
```

Output:

```
Index out of bounds!
```

Explanation:

- Instead of using a try-catch block to handle an `ArrayIndexOutOfBoundsException`, we use an if-else statement to check the index bounds.

- This is more efficient and clearer for predictable conditions.

3. User-Defined Exceptions

Java allows us to create our own exceptions by extending the `Exception` class. This is useful for creating specific exception types for our application.

Lab 3: User-Defined Exception

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

public class Main {
    static void validateAge(int age) {
        try {
            if (age < 18) {
                throw new InvalidAgeException("Age must be 18 or above.");//set
message
            } else {
                System.out.println("Age is valid.");
            }
        } catch (InvalidAgeException e) {//exception won't propagate to the
caller(main method) since we didn't rethrow
            System.out.println("Caught an exception: " + e.getMessage());//get message
        }
    }

    public static void main(String[] args) {
        validateAge(15); // This will internally handle the exception->does not have
access to the exception e
    }
}
```

Output:

```
Caught an exception: Age must be 18 or above.
```

Explanation:

- The `validateAge` method checks if the age is less than 18 and throws a custom checked exception, `InvalidAgeException`, with the message `"Age must be 18 or above."`.
- The exception is caught within the `validateAge` method itself, and the message is printed using `e.getMessage()`. Since the exception is not rethrown, it does not propagate to the caller (`main` method).
- In the `main` method, the `validateAge(15)` call executes, but the `main` method has no direct access to the exception because it is fully handled inside `validateAge`.

Lab 4: User-Defined Exception using throws

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

public class Main {
    // Declare that this method may throw an InvalidAgeException
    static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above."); // Throw the
exception
        } else {
            System.out.println("Age is valid.");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(15); // Call the method that may throw an exception
        } catch (InvalidAgeException e) { // Catch the propagated exception
            System.out.println("Caught an exception: " + e.getMessage()); // Handle
the exception
        }
    }
}
```

Sample Output:

```
Caught an exception: Age must be 18 or above.
```

---

4. Catching Exception: try, catch

The `try` block contains code that might throw an exception. The `catch` block contains code that handles the exception.

Lab 5: Catching Exception

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};//lower bound=0 and upperbound=2 so 0,1 and 2
are in bounds/boundaries
            System.out.println(numbers[5]); // This will cause an
ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds!");
        }
    }
}
```

Output:

```
Array index is out of bounds!
```

Explanation:

- The `try` block attempts to access an index that is out of bounds, causing an `ArrayIndexOutOfBoundsException`.
- The `catch` block catches the exception and prints an error message.

5. Throwing and Rethrowing: throw, throws

- `throw`: Used to explicitly throw an exception.
- `throws`: Used in method signatures to declare that a method might throw one or more exceptions.(Lab 4)

Lab 6: Throwing and Rethrowing

```
public class RethrowExample {
    public static void checkAge(int age) {
        try {
            if (age < 18) {
                throw new ArithmeticException("Access denied - You must be at least 18
years old.");
            } else {
                System.out.println("Access granted - You are old enough!");
            }
        } catch (ArithmeticException e) {
            System.out.println("Initial exception caught: " + e.getMessage());
            throw e;// Rethrow the exception in catch block to propagate it further-
>the exception is rethrown to propagate it to the caller (main method).
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15); // This will throw and rethrow an ArithmeticException
        } catch (ArithmeticException e) {// Catch the rethrown exception in the main
method
            System.out.println("Final exception caught in main: " + e.getMessage());
        }
    }
}
```

Output:

```
Initial exception caught: Access denied - You must be at least 18 years old.
Final exception caught in main: Access denied - You must be at least 18 years old.
```

Explanation:

- The `checkAge` method throws an `ArithmeticException` if the age is less than 18.
- The exception is caught in the `checkAge` method, logged with an initial message, and then rethrown using `throw e;` to propagate it further.
- In the `main` method, the rethrown exception is caught and handled, printing a final error message.

---

6. Cleaning Up Using the finally Clause

The `finally` block is used to execute important code such as closing resources, regardless of whether an exception is thrown or not.

Lab 7: finally Clause

```
import java.util.Scanner;

public class TextInputExample {
    public static void main(String[] args) {
        Scanner scanner = null; // Declare Scanner outside the try block

        try {
            scanner = new Scanner(System.in); // Initialize Scanner to read from
keyboard
            System.out.println("Enter some text:");

            String input = scanner.nextLine(); // Read a single line of input
            System.out.println("You entered: " + input); // Display the input
        } catch (Exception e) { // Catch any unexpected exceptions
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
            // Ensure the Scanner is closed in the finally block
            if (scanner != null) {
                scanner.close(); // Close the Scanner to release resources
                System.out.println("Scanner closed successfully.");
            }
        }
    }
}
```

Sample Output

```
Enter some text:
Hello, World!
You entered: Hello, World!
Scanner closed successfully.
```

Best Practices

1. **Use Try-With-Resources**:
    - Starting from Java 7, we can use the **try-with-resources** syntax to automatically close resources like `Scanner`. This eliminates the need for an explicit `finally` block.

      Lab 8: Try-With-Resources

      ```
      import java.util.Scanner;
      public class TextInputExample {
           public static void main(String[] args) {
               try (Scanner scanner = new Scanner(System.in)) { // Initialize
      ```

```
Scanner within try-with-resources
            System.out.println("Enter some text:");
            String input = scanner.nextLine(); // Read a single line of
input
            System.out.println("You entered: " + input); // Display the
input
        } catch (Exception e) { // Catch any unexpected exceptions
            System.out.println("An error occurred: " + e.getMessage());
        }finally {
            System.out.println("Scanner closed successfully."); // This
message will still be printed
         }
    }
}
```

Sample Output

```
Enter some text:
Hello, World!
You entered: Hello, World!
Scanner closed successfully.
```

2. **Always Close Resources**:
    - Whether using `finally` or try-with-resources, ensure that all resources are properly closed to avoid memory leaks.
3. **Handle Exceptions Gracefully**:
    - Provide meaningful error messages to help debug issues.

Summary

- **Basic Exceptions**: Handle unexpected errors using try-catch blocks.
- **Proper Use of Exceptions**: Use exceptions for exceptional conditions, not for regular control flow.
- **User-Defined Exceptions**: Create custom exceptions by extending the `Exception` class.
- **Catching Exceptions**: Use try-catch blocks to handle exceptions.
- **Throwing and Rethrowing**: Use `throw` to throw exceptions and `throws` to declare them in method signatures.
- **finally Clause**: Use the `finally` block to execute cleanup code that must run regardless of whether an exception occurs.
