Adv Java 04 - Exception Handling

Exception handling is a critical aspect of programming in Java. It allows developers to manage and respond to unexpected errors that may occur during program execution. In this tutorial, we'll cover the basics of exception handling in Java for beginners.

Agenda

1. Introduction to Exceptions

An exception is an event that disrupts the normal flow of a program. When an exceptional situation occurs, an object representing the exception is thrown. Exception handling allows you to catch and handle these exceptions, preventing your program from crashing.

2. Types of Exceptions

In Java, exceptions are broadly categorized into two types: checked exceptions and unchecked exceptions.

Checked Exceptions

These are checked at compile-time, and the programmer is required to handle them explicitly using try-catch blocks or declare them in the method signature using the throws keyword.

Checked exceptions extend the Exception class (directly or indirectly) but do not extend RuntimeException. They are subject to the compile-time checking by the Java compiler, meaning the compiler ensures that these exceptions are either caught or declared.

Some common examples of checked exceptions include:

- IOException
- SQLException
- ClassNotFoundException
- InterruptedException

Handling checked exceptions involves taking appropriate actions to address the exceptional conditions that may arise during program execution. There are two primary ways to handle checked exceptions: using the try-catch block and the throws clause.

The try-catch block is used to catch and handle exceptions. When a block of code is placed inside a try block, any exceptions that occur within that block are caught and processed by the corresponding catch block.

Example

```java
import java.io.FileNotFoundException;
import java.io.FileReader;

public class FileReaderMethodExample {
    public static void main(String[] args) {
        try {
            readFile("example.txt");
        } catch (FileNotFoundException e) {
            System.err.println("FileNotFoundException: " + e.getMessage());
        }
    }

    // Method with a throws clause
    static void readFile(String fileName) throws FileNotFoundException {
        FileReader fileReader = new FileReader(fileName);
        // Code to read from the file
    }
}
```

Unchecked Exceptions

These are not checked at compile-time, and they are subclasses of RuntimeException. They usually indicate programming errors, and it's not mandatory to handle them explicitly. Unchecked exceptions also known as runtime exceptions, are exceptions that occur during the execution of a program.

Unchecked exceptions can occur at runtime due to unexpected conditions, such as division by zero, accessing an array index out of bounds, or trying to cast an object to an incompatible type.

Some common examples of unchecked exceptions include:

`ArithmeticException` : Occurs when an arithmetic operation encounters an exceptional condition, such as division by zero.

`NullPointerException` : Occurs when trying to access a member (field or method) on an object that is null.

`ArrayIndexOutOfBoundsException` : Occurs when trying to access an array element with an index that is outside the bounds of the array.

`ClassCastException` : Occurs when attempting to cast an object to a type that is not compatible with its actual type.

```java
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0); // This may throw an ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    static int divide(int a, int b) {
        return a / b;
    }
}
```

In this example, the divide method may throw an ArithmeticException if the divisor b is zero. The try-catch block catches the exception and handles it, preventing the program from terminating abruptly.

3. Handling Exceptions

The try-catch Block

The try-catch block is used to handle exceptions. The code that might throw an exception is placed inside the try block, and the code to handle the exception is placed inside the catch block.

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

Example:

```java
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0); // This may throw an ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    static int divide(int a, int b) {
        return a / b;
    }
}
```

Multiple catch Blocks

You can have multiple catch blocks to handle different types of exceptions that may occur within the try block.

```java
try {
    // Code that might throw an exception
    // ...
} catch (ExceptionType1 e1) {
    // Code to handle ExceptionType1
    // ...
} catch (ExceptionType2 e2) {
    // Code to handle ExceptionType2
    // ...
}
```

Example:

```java
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length()); // This may throw a NullPointerException
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException: " + e.getMessage());
        } catch (NullPointerException e) {
            System.out.println("NullPointerException: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Generic Exception: " + e.getMessage());
        }
    }
}
```

The finally Block

The finally block contains code that will be executed regardless of whether an exception is thrown or not. It is often used for cleanup operations, such as closing resources.

```java
try {
    // Code that might throw an exception
    // ...
} catch (ExceptionType e) {
    // Code to handle the exception
    // ...
} finally {
    // Code that will be executed regardless of exceptions
    // ...
}
```

Example:

```
public class FinallyBlockExample {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            int result = divide(10, 2);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException: " + e.getMessage());
        } finally {
            System.out.println("Inside finally block");
        }
    }

    static int divide(int a, int b) {
        return a / b;
    }
}
```

Throwing Exceptions

You can use the throw keyword to explicitly throw an exception in your code. This is useful when you want to signal an exceptional condition.

```
public ReturnType methodName() throws ExceptionType1, ExceptionType2 {
    // Method implementation
    ...
    if(condition){
         throw new ExceptionType1("Error message");
    }
    ...
}
```

The `throws` clause is used in a method signature to declare that the method may throw checked exceptions. It informs the caller that the method might encounter certain exceptional conditions, and the caller is responsible for handling these exceptions.

Example

```
public class ThrowExample {
    public static void main(String[] args) {
        try {
            validateAge(15); // This may throw an InvalidAgeException
        } catch (InvalidAgeException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
    static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older");
        }
        System.out.println("Valid age");
    }
}

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

5. Custom Exceptions

You can create your own custom exceptions by extending the Exception class or one of its subclasses.

Example:

```
public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new CustomException("Custom exception message");
        } catch (CustomException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
        }
    }
}

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

6. Best Practices

- Catch specific exceptions rather than using a generic catch (Exception e) block whenever possible.
- Handle exceptions at an appropriate level in your application. Don't catch exceptions if you can't handle them effectively.
- Clean up resources (e.g., closing files or database connections) in the finally block.
- Log exceptions or relevant information to aid in debugging.

Best Practices for Checked Exceptions

- Handle or Declare: Always handle checked exceptions using the try-catch block or declare them in the method signature using the throws clause.

- Provide Meaningful Messages: When catching or throwing checked exceptions, include meaningful messages to aid in debugging.
- Close Resources in a finally Block: If a method opens resources (e.g., files or database connections), close them in a finally block to ensure proper resource management.

Best Practices for Handling Unchecked Exceptions

- Use Defensive Programming: Validate inputs and conditions to avoid common causes of unchecked exceptions.
- Catch Specific Exceptions: When using a try-catch block, catch specific exceptions rather than using a generic catch (RuntimeException e) block. This allows for more targeted handling.
- Avoid Suppressing Exceptions: Avoid using empty catch blocks that suppress exceptions without any meaningful action. Log or handle exceptions appropriately.
- Logging: Consider logging exceptions using logging frameworks (e.g., SLF4J) to record information that can aid in debugging.

Conclusion

Exception handling is a crucial aspect of Java programming, allowing developers to gracefully handle unexpected errors and improve the robustness of their applications. By understanding the basics of exception handling and following best practices, you can write more resilient and reliable Java code. As you gain experience, you'll become proficient in anticipating and addressing potential issues in your programs

--- End ---