**Unit-5 (Exception Handling)**

The **Exception Handling in Java** is one of the powerful **mechanism to handle the runtime error**s so that the normal flow of the application can be maintained.. Java Exception Handling is a mechanism to handle runtime errors such as **ClassNotFoundException, IOException, SQLException, NullPointerException**, etc.

When an exception occurs within a method, it creates an **object**. This **object** is called the **exception object**. It contains information about the exception, such as the **name** and **description** of the exception and the **state of the program** when the exception occurred.

**Major reasons why an exception Occurs**

* Invalid user inputs
* Device failure
* Loss of network connection
* Physical limitations (out-of-disk memory)
* Code errors
* Opening an unavailable file

**Errors** represent **irrecoverable** **conditions** such as Java virtual machine (JVM) running **out of memory**, **memory leaks**, **stack overflow errors**, **library incompatibility**, **infinite recursion**, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

**Difference between exception and error**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Errors** | **Exceptions** |
| **1.** | The error indicates trouble that primarily occurs due to the lack of system resources. | The exceptions are the issues that can appear at runtime and compile time. |
| **2.** | It is not possible to recover from an error. | It is possible to recover from an exception. |
| **3.** | In java, all the errors are unchecked. | In java, the exceptions can be both checked and unchecked. |
| **4.** | The system in which the program is running is responsible for errors. | The code of the program is responsible for exceptions. |
| **5.** | They are described in the **java.lang.Error** package. | They are described in **java.lang.Exception** package |

In Java, exception handling is done using the "try-catch" blocks. Here's how you can catch and handle exceptions in Java:

1. **Try block:** The code that may throw an exception is placed within a "try" block. It is enclosed within curly braces.

try {

// Code that may throw an exception

}

2. **Catch block(s):** Following the try block, one or more "catch" blocks can be added to handle specific types of exceptions. Each catch block specifies the type of exception it can handle.

try {  
 // Code that may throw an exception

} catch (ExceptionType1 e) {  
 // Exception handling for ExceptionType1

} catch (ExceptionType2 e) {  
 // Exception handling for ExceptionType2

}

You can have multiple catch blocks for different exception types. When an exception occurs, the catch blocks are evaluated sequentially to find a matching catch block. If a matching catch block is found, the code within that catch block is executed.

**3. Handling the exception**: Inside each catch block, you can include code to handle the exception. This could involve displaying an **error message**, **logging the exception**, **performing error recovery**, or any other appropriate action.

try {

// Code that may throw an exception

// ...

} catch (ExceptionType1 e) {

// Exception handling for ExceptionType1

// e.printStackTrace(); // Print stack trace (optional)

} catch (ExceptionType2 e) {

// Exception handling for ExceptionType2

}

4. **Optional finally block:** After the catch block(s), you can include an optional "finally" block. The code within the finally block is executed regardless of whether an exception occurred or not. It is typically used for **cleanup operations**, such as releasing resources like **open files** or **closing database connections**.

try {

// Code that may throw an exception

} catch (ExceptionType1 e) {

// Exception handling for ExceptionType1

} catch (ExceptionType2 e) {

// Exception handling for ExceptionType2

} finally {

// Cleanup code (optional)

}

The finally block will always execute, even if an exception is thrown and caught or if an exception is not thrown at all.

By catching and handling exceptions in Java, you can prevent your program from crashing, provide appropriate error handling, and ensure proper resource management.

**throw Statement:**

The throw keyword is used to explicitly throw an exception from within a method or block of code.

It is used when an exceptional condition occurs that the method cannot handle itself.

throw ex.getMessage();

|  |  |
| --- | --- |
| **Throw Keyword** | **Throws Keyword** |
| You use the Throw keyword to throw an exception explicitly. | You can use the Throws keyword to declare that a method might throw an exception. |
| The Throw keyword is used inside a method. | The Throws keyword is used in the method signature. |
| An instance of exception to be thrown. | Class names of exceptions to be thrown. |
| Cannot handle checked exception | Handle Checked Exception |
| Only one exception | Multiple exceptions |

## Explain exception handling mechanism in java with example

## The key components of Java's exception handling mechanism include:

## try: The try block encloses the code that may throw exceptions.

## catch: The catch block follows the try block and catches specific types of exceptions that occur within the try block. Each catch block handles a particular type of exception.

## finally: The finally block is optional and follows the try-catch blocks. It always executes, whether an exception is thrown or not. It's typically used to release resources, such as closing file handles or database connections.

## throw: The throw keyword is used to explicitly throw an exception within a method.

## throws: The throws keyword is used in method signatures to indicate that the method may throw certain types of exceptions. It delegates the responsibility of handling exceptions to the caller of the method.

import java.util.Scanner;

public class App {

    public static void main(String[] args) throws Exception {

        Scanner scanner = new Scanner(System.in);

        try {

            System.out.print("Enter First Digit: ");

            int a = scanner.nextInt();

            System.out.print("Enter Second Digit: ");

            int b = scanner.nextInt();

            if (b == 0) {

                // Throwing an arithmetic exception if the divisor is zero

                throw new ArithmeticException("Cannot divide by zero.");

            }

            // Attempting to divide numbers

            int result= a / b;

            System.out.println(result);

        } catch (ArithmeticException e) {

            // Handling arithmetic exception

            System.err.println("Error: " + e.getMessage());

        } finally {

            // This block always executes, whether an exception occurred or not

            System.out.println("Finally block executed.");

            scanner.close();

        }

    }

}

Output:

Enter First Digit: 10

Enter Second Digit: 0

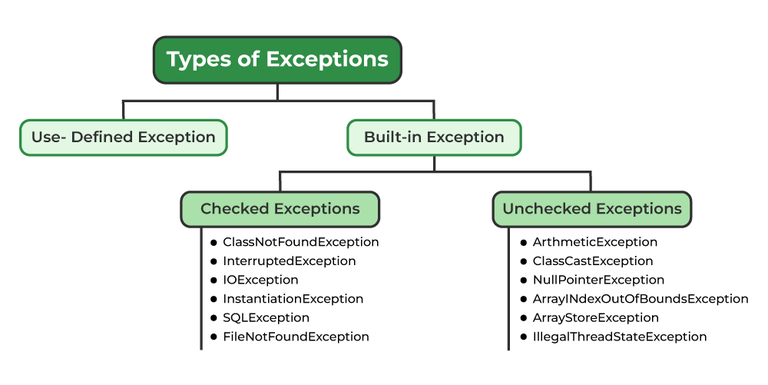
Error: Cannot divide by zero.

Finally block executed.

**Lab:Write a java program for handling exceptions including divide by zero situation using try, catch, finally, throws, and throw in Java.**

## Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



**Difference between Checked and Unchecked Exception:**

* **Checked Exceptions:**Checked exceptions are called **compile-time** exceptions because these exceptions are checked at **compile-time** by the compiler.
* **Unchecked Exceptions:**Checked exceptions are called **runtime-time** exceptions because these exceptions are checked at **runtime-time**.

**Why we need to deal with exception?**

Exception handling is a critical aspect of programming, and there are several reasons why we need to deal with exceptions:

1. **Error detection and reporting**: Exceptions allow us to identify and handle errors or exceptional situations that may occur during program execution. Instead of the program crashing abruptly, exceptions provide a mechanism to detect errors and report them in a controlled manner.

2. **Program stability and robustness**: Exception handling helps in creating stable and robust software. By catching and handling exceptions, we can prevent the program from terminating unexpectedly and provide graceful error recovery mechanisms. This ensures that the program continues to execute and doesn't leave the system in an inconsistent state.

3. **Separation of concerns:** Exception handling allows for the separation of **normal program logic** from **error handling code**. By handling exceptions separately, we can keep the main code clean and focused on its primary purpose, **improving code readability** and **maintainability**.

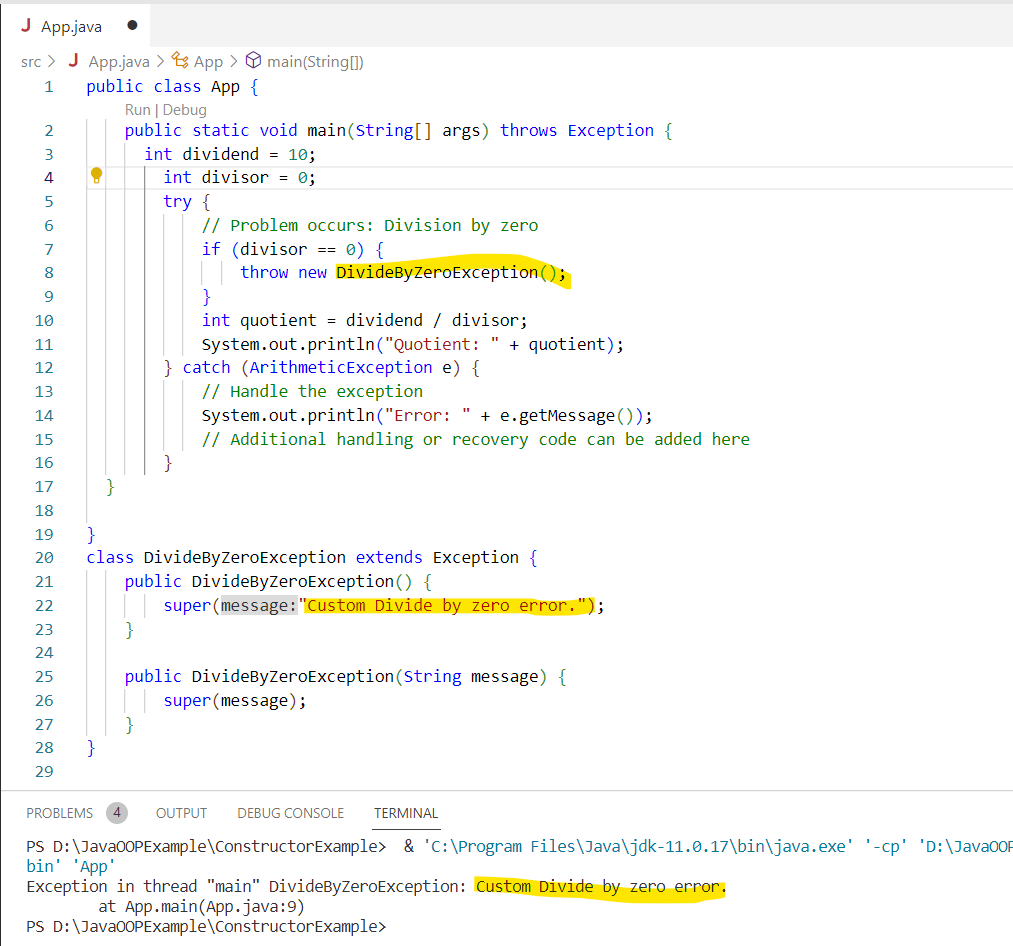
4. **User-friendly experience:** Exception handling enables us to provide meaningful error messages or feedback to the users. Instead of displaying mysterious error codes or crashing, we can catch exceptions and present user-friendly messages, guiding users on how to resolve or recover from errors.

5. **Debugging and troubleshooting:** Exceptions can provide valuable information for debugging and troubleshooting purposes. This information can be used to identify the **root cause of the issue** and fix the problem effectively.

6. **Resource management:** Exception handling plays a crucial role in proper resource management. **For example**, if a file is opened for reading and an exception occurs while reading from it, the exception handling code can ensure that the file is closed properly, preventing resource leaks.

**User Defined Exception:**

We can also define our own exceptions. These are called Custom exceptions or user-defined exceptions.  
Using custom exceptions, we can define our exceptions as per our needs. we can create our own In order to create custom exception, we need to extend **Exception** class that belongs to **java.lang** package.



**Lab:Write a Java program that will read balance and withdraw amount from keyboard and display the remaining balance on screen of balance is greater than withdraw amount otherwise throw an User defined exception with appropriate message.**

import java.util.Scanner;

public class App {

    public static void main(String[] args) throws Exception {

        Scanner scanner = new Scanner(System.in);

        try {

            System.out.print("Enter your balance: ");

            double balance = scanner.nextDouble();

            System.out.print("Enter withdrawal amount: ");

            double withdrawal = scanner.nextDouble();

            // Check if withdrawal amount is valid

            if (withdrawal > balance) {

               throw new MyCustomException("Insufficient balance");

            }

            double remBalance = balance - withdrawal;

            System.out.println("Remaining balance: " + remBalance);

        } catch (MyCustomException ex) {

           System.out.println("Error: " + ex.getMessage());

        } catch (Exception ex) {

           System.out.println("An error occurred: " + ex.getMessage());

        } finally {

            scanner.close();

        }

    }

}

class MyCustomException extends Exception {

    public MyCustomException(String message) {

        super(message);

    }

}

**Output:**

**Enter your balance: 5000**

**Enter withdrawal amount: 6000**

**Error: Insufficient balance**