Exception Handling in Java

Exception handling is a crucial aspect of programming that helps manage errors and unexpected events in a controlled way. Here's a breakdown of the basics and types of exceptions in Java.

Basics of Exception Handling

Exception handling in Java is primarily done using ***try***, ***catch***, and ***finally*** blocks.

1. What is an Exception?

- An exception is an event that disrupts the normal flow of a program. It indicates that an unusual condition has occurred.

2. Why Use Exception Handling?

- To manage errors gracefully without crashing the program.
- To separate error-handling code from regular code, improving readability and maintainability.
- To allow a program to recover from errors and continue executing.

3. Core Components:

try block : This contains the code that might throw an exception.

catch block : This catches the specific exception and allows you to handle it. Contains the code that handles the exception if it occurs.

finally block : This block executes after the try and catch blocks, regardless of whether an exception occurred or not. It's typically used for cleanup activities, like closing files or releasing

throw : Used to explicitly throw an exception.

throws : Declares that a method can throw exceptions.

Types of Exceptions

Exceptions in Java can be categorized into two main types:

1. Checked Exceptions:

These exceptions are checked at compile-time. The compiler requires that you handle these exceptions, either with a `try-catch` block or by declaring them with the throws keyword.

Examples:

- IOException
- SQLException
- ClassNotFoundException

Usage : Typically used for conditions that a reasonable application might want to catch, like file operations, network issues, etc.

2. Unchecked Exceptions:

These exceptions are not checked at compile-time, meaning the compiler does not require you to handle them. They are usually a result of programming errors.

Examples:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException

Usage : Typically indicate bugs or errors in the program, such as accessing an array out of bounds or dividing by zero.

Exception Hierarchy

- The root class for exceptions is Throwable.
- Error: Represents serious problems that a reasonable application should not try to catch. Examples include OutOfMemoryError, StackOverflowError.
- Exception: The base class for all exceptions that applications might want to catch.
- Checked Exceptions: Subclasses of `Exception` (excluding RuntimeException and Error).
- Unchecked Exceptions: Subclasses of RuntimeException.

Simple structure demonstrating exception handling:

```
import java.lang.*;
import java.util.*;

public class ExceptionDemo
{
    public static void main(String[] args)
    {
        try
        {
            // Code that may throw an exception
            int result = 10 / 0; // This will throw ArithmeticException
        }
        catch (ArithmeticException e)
        {
            // Handling the exception
            System.out.println("Error: " + e.getMessage());
        }
        finally
       {
            // Cleanup code (optional)
            System.out.println("Execution completed.");
        }
    }
}
```

You can define your own exceptions by extending the Exception class.
Here’s an example that demonstrates how to create and use multiple user-defined exceptions.

Example: User-Defined Exceptions

Let's create a simple banking application where we handle two custom exceptions: InsufficientFundsException and NegativeAmountException.

```
import java.lang.*;
import java.util.*;


// Custom Exception for Insufficient Funds
class InsufficientFundsException extends Exception
{
   public InsufficientFundsException(String message)
   {
      super(message);
   }
}


// Custom Exception for Negative Amount
class NegativeAmountException extends Exception
{
   public NegativeAmountException(String message)
   {
      super(message);
   }
}
```

```
// Bank Account class
class BankAccount
{
   private double balance;

   public BankAccount(double initialBalance)
   {
      if (initialBalance < 0)
      {
         throw new IllegalArgumentException("Initial balance cannot be negative.");
      }
      this.balance = initialBalance;
   }

   public void deposit(double amount) throws NegativeAmountException
   {
      if (amount < 0)
      {
         throw new NegativeAmountException("Deposit amount cannot be negative.");
      }
      balance += amount;
      System.out.println("\n Deposited: " + amount);
   }
         public void withdraw(double amount) throws InsufficientFundsException,
         NegativeAmountException
         {
                  if (amount < 0)
                  {
                           throw new NegativeAmountException("Withdrawal amount cannot be negative.");
                  }
                  if (amount > balance)
                  {
                           throw new InsufficientFundsException("Insufficient funds for this withdrawal.");
                  }
                  balance -= amount;
                  System.out.println("\n Withdrew: " + amount);
         }

          public double getBalance()
         {
                  return balance;
         }
}
```

```
// Main class
public class BankApp
{
        public static void main(String[] args)
        {
                BankAccount account = new BankAccount(1000);

                 try
                {
                        account.deposit(500);
                        account.withdraw(200);
                        account.withdraw(1500); // This will cause InsufficientFundsException
                }
                catch (InsufficientFundsException | NegativeAmountException e)
                {
                        System.out.println("\n Exception: " + e.getMessage());
                }


                try
                {
                        account.deposit(-100); // This will cause NegativeAmountException
                }
                catch (NegativeAmountException e)
                {
                        System.out.println("Exception: " + e.getMessage());
                }

                System.out.println("\n Current Balance : " + account.getBalance());
        }
}
```

Output:

When you run the program, it will display:

```
        Deposited: 500.0

        Withdrew: 200.0

        Exception: Insufficient funds for this withdrawal.

        Exception: Withdrawal amount cannot be negative.

        Current Balance: 1300.0
```

Explaination:

1. Custom Exceptions:

- InsufficientFundsException : Thrown when a withdrawal amount exceeds the current balance.

- NegativeAmountException: Thrown when a negative deposit or withdrawal amount is attempted.

2. BankAccount Class:

- Contains methods for *deposit* and *withdraw*, each throwing the appropriate exceptions based on the conditions.

3. Main Class (BankApp):

- Creates a *BankAccount* instance and performs various operations, handling the exceptions accordingly.

This demonstrates how to define and use multiple user-defined exceptions in Java effectively.