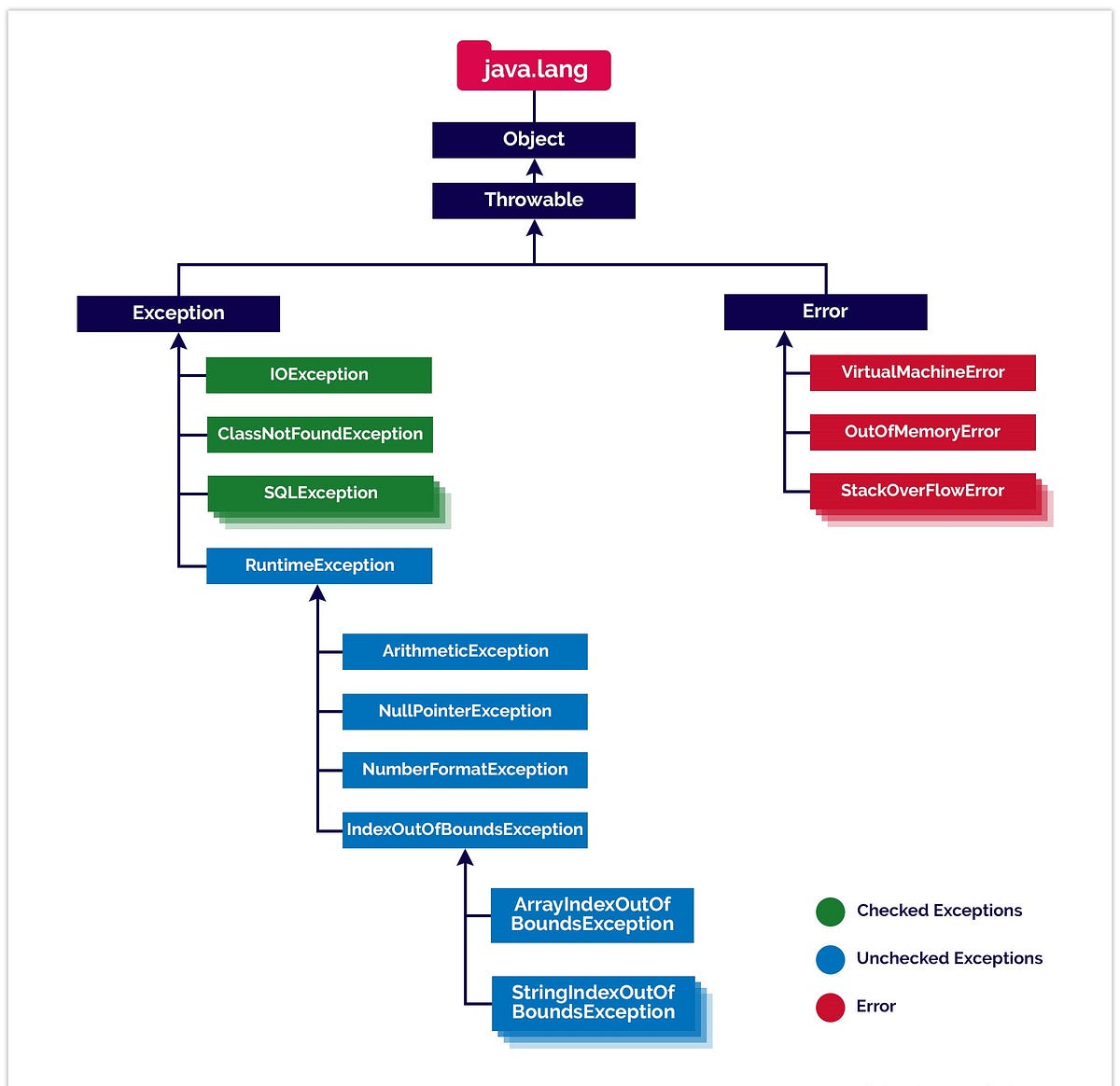
**Exception Handling**

1. **Introduction**

Exception handling is a powerful mechanism that allows a program to deal with runtime errors, making it more robust and error-resistant. It helps separate error-handling code from regular code, allowing the program to continue running even after an error occurs. In Java, exceptions are handled using a set of keywords: try, catch, throw, throws, and finally.

In Java, all exceptions and errors are derived from the Throwable class. The two main branches of the exception hierarchy are Exception and Error. While exceptions represent problems that can be handled by the program, errors typically indicate serious issues that a program cannot recover from.



1. **Types of Exceptions**
2. **Checked Exceptions**: These exceptions are checked at compile-time. If a method might throw a checked exception, it must either handle it using a try-catch block or declare it using the throws keyword.
   * Example: IOException, SQLException
3. **Unchecked Exceptions**: These exceptions are not checked at compile-time, but occur during the execution of the program (Runtime Exception). These exceptions typically occur due to programming bugs.
   * Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException
4. **Errors**: These are serious problems that a program cannot typically recover from. They are not meant to be caught.
   * Example: OutOfMemoryError, StackOverflowError

**3.Exception Handling Keywords**

Let’s take a closer look at the key exception handling keywords in Java.

**try**

The try block is used to wrap code that may throw an exception. If an exception occurs, it is caught in the corresponding catch block.

try {

// Code that might throw an exception

}

**catch**

The catch block handles exceptions thrown in the try block. It specifies the type of exception to handle.

catch (ExceptionType e) {

// Code to handle the exception

}

**finally**

The finally block always executes after the try block, regardless of whether an exception was thrown or not. It is used for cleanup activities like closing files or releasing resources.

finally {

// Cleanup code, always executed

}

**throw**

The throw keyword is used to explicitly throw an exception in your code. It can be used to create custom exceptions or rethrow existing exceptions.

throw new SomeException("Message");

**throws**

The throws keyword is used in the method signature to declare that a method may throw one or more exceptions. This allows the caller of the method to handle the exception.

public void someMethod() throws IOException {

// Code that might throw an IOException

}

**Basic Syntax**

**Syntax of try-catch-finally**

try {

// Code that might throw an exception

} catch (ExceptionType1 e1) {

// Code to handle exception of type ExceptionType1

} catch (ExceptionType2 e2) {

// Code to handle exception of type ExceptionType2

} finally {

// Code that always executes after the try block, regardless of exception

}

**4. Exception Handling Example Programs**

**1. Basic Example with try-catch**

This example demonstrates basic exception handling using a try-catch block.

public class BasicExceptionHandling {

public static void main(String[] args) {

try {

int[] numbers = new int[5];

numbers[10] = 50; // This will cause ArrayIndexOutOfBoundsException

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Exception caught: " + e);

}

System.out.println("Program continues after exception.");

}

}

**Explanation**:

* The program tries to access an element at index 10 of an array that only has 5 elements.
* This causes an ArrayIndexOutOfBoundsException.
* The exception is caught in the catch block, and the message is printed. The program continues executing after the exception is handled.

**Output**:

Exception caught: java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 5

Program continues after exception.

**2. Multiple catch Blocks Example**

Here, multiple exceptions are handled using multiple catch blocks.

public class MultipleCatchExample {

public static void main(String[] args) {

try {

int num = 10;

int result = num / 0; // This will cause ArithmeticException

String str = null;

int length = str.length(); // This will cause NullPointerException

} catch (ArithmeticException e) {

System.out.println("ArithmeticException caught: " + e);

} catch (NullPointerException e) {

System.out.println("NullPointerException caught: " + e);

} catch (Exception e) {

System.out.println("General exception caught: " + e);

}

}

}

**Explanation**:

* First, the division by zero causes an ArithmeticException, which is handled by the first catch block.
* After that, the null pointer exception is thrown when attempting to access str.length(), which is caught by the second catch block.
* The third catch block catches any other general exceptions, but it won't be triggered because we have specific catch blocks for the exceptions thrown.

**Output**:

ArithmeticException caught: java.lang.ArithmeticException: / by zero

NullPointerException caught: java.lang.NullPointerException: Cannot read the length of a null array

**3. Using finally Block**

The finally block is executed no matter what, whether or not an exception is thrown.

public class FinallyBlockExample {

public static void main(String[] args) {

try {

int num = 10;

int result = num / 2; // This will not cause an exception

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e);

} finally {

System.out.println("This block will always execute.");

}

}

}

**Explanation**:

* The program executes normally and does not throw an exception.
* The finally block executes regardless of the success or failure of the try block.

**Output**:

Result: 5

This block will always execute.

**4. Throwing an Exception Explicitly**

You can explicitly throw an exception using the throw keyword.

public class ThrowingExceptionExample {

public static void main(String[] args) {

try {

checkAge(15);

} catch (IllegalArgumentException e) {

System.out.println("Exception caught: " + e);

}

}

static void checkAge(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age must be 18 or older.");

} else {

System.out.println("Age is valid.");

}

}

}

**Explanation**:

* The checkAge method checks if the age is less than 18.
* If so, it throws an IllegalArgumentException.
* The exception is caught in the catch block and the message is printed.

**Output**:

Exception caught: java.lang.IllegalArgumentException: Age must be 18 or older.

**5. Declaring Exceptions with throws**

The throws keyword is used in method signatures to declare that a method might throw one or more exceptions.

import java.io.\*;

public class ThrowsExample {

public static void main(String[] args) {

try {

readFile("nonexistentfile.txt");

} catch (IOException e) {

System.out.println("Exception caught: " + e);

}

}

static void readFile(String filename) throws IOException {

FileReader file = new FileReader(filename); // This might throw IOException

BufferedReader fileInput = new BufferedReader(file);

System.out.println(fileInput.readLine());

}

}

**Explanation**:

* The readFile method declares that it may throw an IOException.
* The IOException is caught in the catch block when the file is not found.

**Output**:

Exception caught: java.io.FileNotFoundException: nonexistentfile.txt (No such file or directory)

**5. Custom Exceptions in Java**

Java allows you to create your own exception classes by extending either the Exception or RuntimeException class. Custom exceptions can be useful to represent domain-specific error conditions.

**Example: Custom Exception**

// Custom checked exception

class InsufficientFundsException extends Exception {

public InsufficientFundsException(String message) {

super(message);

}

}

// Custom unchecked exception

class InvalidAmountException extends RuntimeException {

public InvalidAmountException(String message) {

super(message);

}

}

public class CustomExceptionExample {

public static void main(String[] args) {

try {

withdraw(150, 100); // Should throw InsufficientFundsException

} catch (InsufficientFundsException e) {

System.out.println("Exception caught: " + e.getMessage());

}

try {

deposit(-50); // Should throw InvalidAmountException

} catch (InvalidAmountException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

static void withdraw(double balance, double amount) throws InsufficientFundsException {

if (amount > balance) {

throw new InsufficientFundsException("Insufficient funds for withdrawal.");

}

System.out.println("Withdrawal successful!");

}

static void deposit(double amount) {

if (amount <= 0) {

throw new InvalidAmountException("Deposit amount must be greater than zero.");

}

System.out.println("Deposit successful!");

}

}

**Explanation:**

* **InsufficientFundsException** is a custom checked exception, meaning it must be declared in the method signature using throws if it is thrown.
* **InvalidAmountException** is a custom unchecked exception, extending RuntimeException.

**Output:**

Exception caught: Insufficient funds for withdrawal.

Exception caught: Deposit amount must be greater than zero.

**6. Try-with-Resources**

Java 7 introduced the **try-with-resources** statement, which simplifies the management of resources like file streams, sockets, etc., by automatically closing them at the end of the try block, even if an exception occurs.

**Example: Try-with-Resources**

import java.io.\*;

public class TryWithResourcesExample {

public static void main(String[] args) {

try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {

String line = reader.readLine();

System.out.println(line);

} catch (IOException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

**Explanation:**

* In this example, the BufferedReader is opened within the try block.
* When the try block finishes, the BufferedReader is automatically closed, even if an exception occurs.

**Key Points:**

* Any class that implements AutoCloseable (like BufferedReader, FileInputStream, etc.) can be used in a try-with-resources statement.
* The finally block is not needed when using try-with-resources because resources are automatically closed.

**7. Catch with Multiple Exceptions**

Java allows you to catch multiple exceptions in a single catch block using the pipe (|) operator. This feature was introduced in Java 7.

**Example: Catch Multiple Exceptions**

public class CatchMultipleExceptions {

public static void main(String[] args) {

try {

int num = 10;

int result = num / 0; // ArithmeticException

String str = null;

str.length(); // NullPointerException

} catch (ArithmeticException | NullPointerException e) {

System.out.println("Exception caught: " + e);

}

}

}

**Explanation:**

* The program attempts two operations: division by zero and accessing the length of a null string.
* Both ArithmeticException and NullPointerException are caught in the same catch block using the pipe (|) operator.

**Output:**

Exception caught: java.lang.ArithmeticException: / by zero

**Key Points:**

* The pipe (|) operator allows handling different exceptions in a single catch block, reducing the need for multiple catch blocks when handling similar exceptions.
* This feature is available from Java 7 and above.

**8. Methods of the Exception Class in Java**

The Exception class in Java provides several methods that help to retrieve information about an exception. Here are some key methods:

1. **getMessage()**
   * Returns a detailed message string describing the exception. This message is provided when the exception is thrown.
2. **getCause()**
   * Returns the cause of the exception, which can be another throwable (i.e., another exception) that caused the current exception.
3. **toString()**
   * Returns a string representation of the exception, which includes the class name and the result of getMessage(). It’s often used for logging and debugging.
4. **printStackTrace()**
   * Prints the stack trace of the exception to the standard error stream. This is useful for debugging and seeing where the exception occurred in the code.

These methods provide important diagnostic information when dealing with exceptions and help in effective error handling and debugging.