**Exception Handling:** Exception Handling is a mechanism in programming that deals with runtime errors, allowing programs to respond to unexpected situations gracefully. Exceptions represent abnormal conditions that can occur during the execution of a program, such as division by zero, array index out of bounds, or attempting to open a file that doesn't exist.

**Try-Catch Block:**

* The try block contains the code that may throw an exception.
* The catch block handles the exception if one occurs in the try block.
* A try block can have multiple catch blocks to handle different types of exceptions.

**try {**

**// Code that may throw an exception**

**} catch (ExceptionType1 e) {**

**// Handle exception type 1**

**} catch (ExceptionType2 e) {**

**// Handle exception type 2**

**}**

Finally Block:

* The finally block contains code that is always executed, whether an exception occurs or not.
* It is typically used for cleanup operations.

try {

// Code that may throw an exception

} catch (ExceptionType e) {

// Handle the exception

} finally {

// Code that always executes

}

Throw Statement:

* The throw statement is used to explicitly throw an exception.
* It is typically used to indicate that a certain condition has occurred that requires special handling.

Example:



**Types:**

Exception handling in Java involves dealing with different types of exceptions. Java divides exceptions into two main categories: **Checked Exceptions** and **Unchecked Exceptions**.

**1. Checked Exceptions:**

Checked exceptions are the exceptions that the Java compiler requires you to handle. They are typically related to external factors that are beyond the control of the program. Checked exceptions must be caught or declared using the throws clause.

Examples of Checked Exceptions:

* IOException
* SQLException
* FileNotFoundException

Example:



**2. Unchecked Exceptions (Runtime Exceptions):**

Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked at compile time. They usually indicate programming errors, such as logical errors or misuse of API methods. Unchecked exceptions do not require explicit handling.

Examples of Unchecked Exceptions:

* ArithmeticException
* NullPointerException
* ArrayIndexOutOfBoundsException

**Example:**

public class UncheckedExceptionExample {

public static void main(String[] args) {

int result;

int divisor = 0;

try {

result = 10 / divisor; // ArithmeticException

} catch (ArithmeticException e) {

// Handle the ArithmeticException

e.printStackTrace();

}

}

}

**Custom Exceptions:**

You can create your own custom exceptions by extending the Exception class (for checked exceptions) or the Runtime Exception class (for unchecked exceptions). This allows you to define exceptions that are specific to your application.



**InvalidAgeException. This exception will be thrown if an attempt is made to create a Person object with an invalid age.**



**Chained Exceptions:** Chained Exceptions in Java allow you to associate one exception with another, providing a way to represent more complex error scenarios. This is useful when you want to give additional context or information about why an exception occurred.



**3. Suppressed Exceptions:**

Java 7 introduced the concept of suppressed exceptions. If an exception is thrown in the try block and another exception is thrown in the finally block, the exception in the finally block is considered suppressed. You can access the suppressed exceptions using the getSuppressed method.



1. **StringBuilder**:
   * Introduced in Java 5.
   * It is not synchronized, meaning it is not thread-safe.
   * Generally faster than **StringBuffer** because it is not synchronized.
   * Recommended to use in single-threaded environments where thread safety is not a concern.
   * Use **StringBuilder** when you need to perform string manipulations in a single-threaded environment for better performance.
2. **StringBuffer**:
   * Available since the early versions of Java.
   * It is synchronized, making it thread-safe.
   * Slower than **StringBuilder** due to the overhead of synchronization.
   * Recommended to use in multi-threaded environments where thread safety is required.
   * Use **StringBuffer** when you need to perform string manipulations in a multi-threaded environment to ensure thread safety.