**EXCEPTION HANDLING**

* **Exception Handling** in Java is a mechanism that allows you to handle runtime errors and maintain the normal flow of an application.

**Types of Exceptions in Java:**

* Checked Exceptions
* Unchecked Exceptions
* Errors

**CHECKED EXCEPTIONS**

**Definition:**

* Checked exceptions are exceptions that are checked at compile-time.
* The compiler ensures that these exceptions are either handled using a try-catch block or declared using the throws keyword in the method signature.

**Common Examples:**

* IOException
* SQLException
* FileNotFoundException

|  |  |
| --- | --- |
| **Exception Type** | **Cause of Creation** |
| IOException | Is thrown when Issues with file access, network communication, etc. |
| SQLException | Is thrown when Problems with SQL queries, connection issues, etc. |
| FileNotFoundException | Is thrown when attempting to open a non-existent file. |
| ClassNotFoundException | Is thrown when the Java run-time system is unable to find the referred class. |
| IllegalAccessException | Is thrown when you refer an object, class, variable, constructor, or a method that is not accessible. |
| InstantiationException | Is thrown when you try to create an instance of a class by using the newInstance() method, but the referred class cannot be instantiated. |
| NoSuchMethodException | Is thrown when a particular method cannot be found. |
| ParseException | Is thrown when Incorrect format or syntax in strings being parsed. |
| InterruptedException | Is thrown when a thread is interrupted while it is waiting, sleeping, or busy. |
| CloneNotSupportedException | Is thrown When an object does not implement the Cloneable interface. |
| NoSuchFieldException | Accessing a field that does not exist in the class definition. |
| InvocationTargetException | An exception occurred during reflection method invocation. |

**UNCHECKED EXCEPTIONS**

**Definition:**

* Unchecked exceptions (also known as **Runtime Exceptions**) are exceptions that are not checked at compile-time but rather at runtime.
* They are the result of programming errors, such as logical mistakes or improper use of APIs.

**Common Examples:**

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException

**ERRORS**

**Definition:**

* Errors are serious issues that occur at runtime and cannot be handled by the typical exception-handling mechanisms.
* They are mostly related to the environment in which the application is running.
* Errors are beyond the control of the program and are usually not recoverable.

**Common Examples:**

* StackOverflowError
* OutOfMemoryError
* VirtualMachineError

|  |  |
| --- | --- |
| **Exception Type** | **Cause of Creation** |
| NullPointerException | Dereferencing a null object or accessing its methods/fields. |
| ArrayIndexOutOfBoundsException | Accessing an array element with an index that is out of bounds. |
| ArrayStoreException | Occurs when you assign an element to an array that is not compatible with type of data that can be stored in that array. |
| ArithmeticException | Division by zero or overflow in calculations. |
| ClassCastException | Invalid type casting between incompatible classes. |
| IllegalArgumentException | Providing invalid arguments to a method (e.g., negative size). |
| IllegalStateException | Calling a method when the object is not in an appropriate state. |
| ConcurrentModificationException | Modifying a collection (e.g., adding/removing elements) during iteration. |
| UnsupportedOperationException | Attempting to modify an unmodifiable collection or object. |
| NumberFormatException | Parsing a string that does not have the appropriate format. |
| NegativeArraySizeException | Occurs when you create an array with a negative size. |

**Exception Hierarchy in Java:**

* **Throwable:** The root class for all exceptions and errors in java.
* **Exception:** Used for recoverable conditions.
* **Checked Exceptions:** Must be handled at compile-time.
* **Unchecked Exceptions(Runtime Exceptions):** Occur at runtime due to programming errors.
* **Error:** Represents serious issues that cannot be handled by the program

**Sample Format:**

java.lang.Object

└── java.lang.Throwable

├── java.lang.Exception

│ ├── java.io.IOException

│ ├── java.sql.SQLException

│ └── java.lang.RuntimeException

│ ├── java.lang.NullPointerException

│ ├── java.lang.ArrayIndexOutOfBoundsException

│ └── java.lang.ArithmeticException

└── java.lang.Error

├── java.lang.StackOverflowError

└── java.lang.OutOfMemoryError

**IMPLEMENTING EXCEPTION HANDLING:**

**Keywords Used in Exception Handling**

* try
* catch
* finally
* throw
* throws

**try**

* Code that might throw an exception is placed inside a try block.

**catch**

* This block handles the exception thrown by the try block.

**finally**

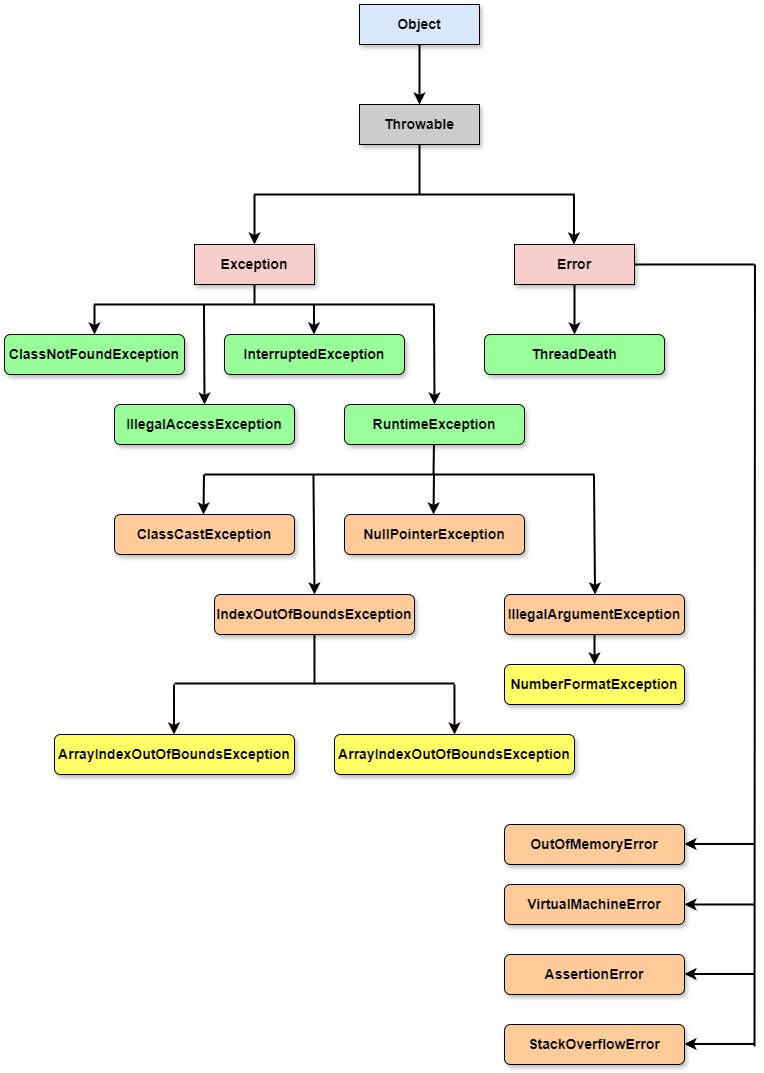
* The finally block always executes, regardless of whether an exception occurs or not.
* It is typically used for cleanup code, such as closing files or releasing resources.

**throw**

* The throw keyword is used to explicitly throw an exception.

**throws**

* The throws keyword is used in the method signature to declare that a method may throw exceptions.



**Using try and catch Blocks:**

* A try block encloses the statements that might raise an exception and define one or more exception handlers associated with it.
* If an exception is raised within the try block, the appropriate exception handler that is associated with the try block process the exception.
* In Java, the catch block is used as an exception handler.
* A try block must have at least one catch block that follows the try block, immediately.
* The catch block specifies the exception type that you need to catch.

**Syntax:**

try

{

//Statement that can cause an exception.

}

catch(exceptionname obj)

{

//Error handling code.

}

* In the preceding syntax, the catch block accepts the object of the Throwable class or its subclass that refers to the exception caught, as a parameter.
* When the exception is caught, the statements within the catch block are executed.

**Example: Without using try and catch block**

package com.demo.JavaCode;  
  
import java.util.Scanner;  
  
public class Addition {  
 public static void main(String[] args){  
 int num1, num2, result;  
 Scanner sc = new Scanner(System.in);  
 System.out.println("Enter the 1st number:");  
 num1 = sc.nextInt();  
 System.out.println("Enter the 2nd number:");  
 num2 = sc.nextInt();  
 result = num1 + num2;  
 System.out.println("The result is: "+result);  
 }  
}

**Example: With using try and catch block**

package com.demo.JavaCode;  
  
import java.util.Scanner;  
  
public class Addition1 {  
 public static void main(String[] args){  
 int num1, num2, result;  
 String snum1, snum2;  
 Scanner sc = new Scanner(System.in);  
 try{  
 System.out.println("Enter the 1st number:");  
 snum1 = sc.next();  
 System.out.println("Enter the 2nd number:");  
 snum2 = sc.next();  
 num1 = Integer.parseInt(snum1);  
 num2 = Integer.parseInt(snum2);  
 result = num1 + num2;  
 System.out.println("The result is: "+result);  
 }catch (Exception e){  
 System.out.println("Enter only numeric values:"+e.getMessage());  
 }  
 }  
}

**Note:**

* The **parseInt()** method is a **static method** defined inside the Integer wrapper class.
* It is used to convert a numeric string value, such as 12, to an integer value.

**Example: NumberFormatException**

package com.demo.JavaCode;  
  
public class TryCatchExample {  
 String abc = "hello";  
  
 public void name(){  
 try {  
 int i = Integer.*parseInt*(abc);  
 System.*out*.println("The value of converted string of i: "+ i);  
 }catch (Exception e){  
 e.printStackTrace();  
 System.*out*.println(e);  
 }  
 }  
 public static void main(String[] args){  
 TryCatchExample tce = new TryCatchExample();  
 tce.name();  
 }  
}

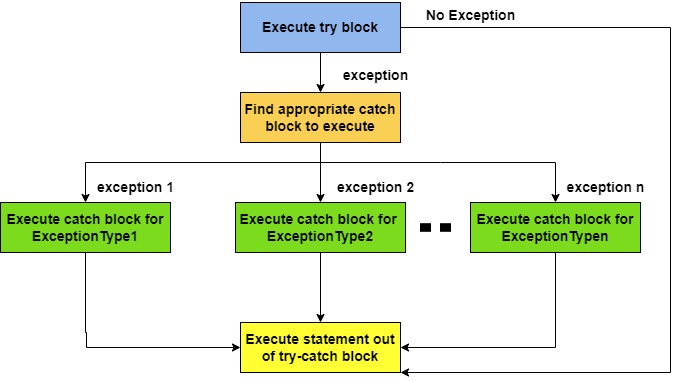
**Java Catch Multiple Exceptions**

**Java multi-catch block.**

* A try block can be followed by one or more catch blocks.
* Each catch block must contain a different exception handler.
* So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

**Points to remember**

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.



**Syntax**

try

{

//statements

}

catch(exceptionname1 obj1)

{

//statements to handle exception

}

catch(exceptionname2 obj2)

{

//statements to handle exception

}

catch(exceptionnameN objN)

{

//statements to handle exception

}

**Example:**

package com.demo.JavaCode;  
  
import java.util.Scanner;  
  
public class Division {  
 public static void main(String[] args) {  
 int num1, num2, result;  
 Scanner sc = new Scanner(System.*in*);  
 try {  
 System.*out*.println("Enter the 1st number: ");  
 num1 = sc.nextInt();  
 System.*out*.println("Enter the 2nd number: ");  
 num2 = sc.nextInt();  
 result = num1 / num2;  
 System.*out*.println("The result is: "+result);  
  
 }catch (ArithmeticException e){  
 System.*out*.println("Division performed by zero...");  
 }catch (Exception e){  
 System.*out*.println("Please input only numeric values...");  
 }  
 }  
}

**Nested try block**

* A nested try block is a structure where you place a try block within another try block.
* Each try can have its own set of catch blocks to handle exceptions specifically.

**Example:**

public class NestedTryExample {

public static void main(String[] args) {

try {

// Outer try block

int[] numbers = {1, 2, 3};

int a = numbers[0];

try {

// Inner try block

int b = numbers[3]; // This will cause an ArrayIndexOutOfBoundsException

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Caught an ArrayIndexOutOfBoundsException in the inner try block: " + e.getMessage());

}

// Additional code in outer try block

System.out.println("Value of a: " + a);

} catch (Exception e) {

System.out.println("Caught an exception in the outer try block: " + e.getMessage());

}

}

}

**Explanation:**

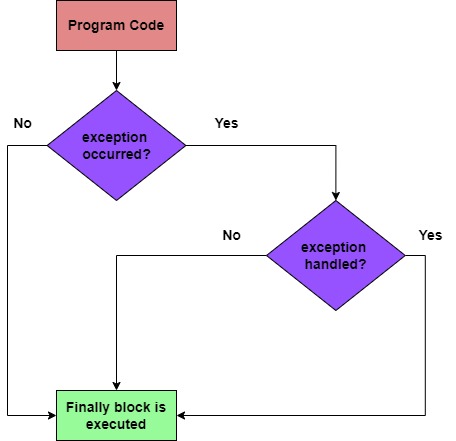
* **Outer Try Block**: This block contains an array and accesses its first element.
* **Inner Try Block**: This attempts to access an index that is out of bounds, leading to an ArrayIndexOutOfBoundsException.
* **Inner Catch Block**: This catches the specific exception from the inner try block and prints an error message.
* **Outer Catch Block**: This would catch any other exceptions not caught by the inner try block.

**Java finally block**

* **Java finally block** is a block used to execute important code such as closing the connection, etc.
* Java finally block is always executed whether an exception is handled or not.
* Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
* The finally block follows the try-catch block.

**Why use java finally block**

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.



**Syntax:**

try

{

//Block of code

}

catch(exceptionname1 obj1)

{

//Statement to handle exception

}

catch(exceptionname1 obj1)

{

//Statement to handle exception

}

finally

{

//Block of code that is always executed irrespective of an exception being raised or not

}

**Example:**

public class FinallyExample {

public static void main(String[] args) {

try {

System.out.println("Trying to divide by zero...");

int result = 10 / 0; // This will cause an ArithmeticException

System.out.println("Result: " + result); // This line won't be executed

} catch (ArithmeticException e) {

System.out.println("Caught an exception: " + e.getMessage());

} finally {

System.out.println("Finally block executed. Cleaning up resources.");

}

System.out.println("Program continues after the try-catch-finally block.");

}

}

**Explanation:**

* **Try Block**: The code that may throw an exception is placed in the try block. Here, we attempt to divide by zero, which will throw an ArithmeticException.
* **Catch Block**: If an exception occurs, control transfers to the catch block where the exception is handled. In this case, it prints an error message.
* **Finally Block**: The finally block is executed after the try and catch blocks, regardless of whether an exception was thrown or caught. It typically contains cleanup code, such as closing files or releasing resources.

**Java throw keyword**

* In Java, the throw statement is used to explicitly throw an exception.
* It can be used to indicate that an error or exceptional condition has occurred in the program, allowing developers to handle the error appropriately.

**Definition**

* **throw Statement**: It is used to throw an exception explicitly, either a built-in exception (like NullPointerException, ArithmeticException, etc.) or a user-defined exception.
* When an exception is thrown, the normal flow of execution is interrupted, and the control is transferred to the nearest catch block that can handle that exception.

**Syntax:**

throw ThrowableObj

**Example: ArithmeticException**

package com.demo.JavaCode;  
  
public class ThrowExample {  
 public static void checkNum(int num){  
 if (num < 1){  
 throw new ArithmeticException("\n Number is negative, cannot calculate square");  
 }else {  
 System.*out*.println("Square of "+ num + " is "+ (num\*num));  
 }  
 }  
 public static void main(String[] args){  
 *checkNum*(-1);  
  
 }  
}

**Example: IllegalArgumentException**

public class ThrowExample {

// Method to validate age

static void validateAge(int age) {

if (age < 18) {

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

} else {

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

}

}

public static void main(String[] args) {

try {

validateAge(15); // This will throw an exception

} catch (IllegalArgumentException e) {

System.out.println("Caught an exception: " + e.getMessage());

}

try {

validateAge(20); // This will not throw an exception

} catch (IllegalArgumentException e) {

System.out.println("Caught an exception: " + e.getMessage());

}

}

}

**Note:**

* An exception can be handled either in the method that raises the exception or in the calling method.
* throw keyword will be activated at the compile time process.

**Java throws keyword**

* The **Java throws keyword** is used to declare an exception.
* It gives an information to the programmer that there may occur an exception.
* So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
* In Java, the throws keyword is used in method declarations to specify that a method can throw one or more exceptions.
* This allows the calling method to handle those exceptions appropriately, providing a way to deal with error conditions without losing control of the program flow.

**Definition:**

**Syntax:**

* The **throws** clause follows the method signature and can specify multiple exceptions separated by commas.

return\_type method\_name() **throws** exception\_class\_name{

//method code

}

**Example:**

package com.demo.JavaCode;

public class ThrowsExample {

public static int divideNum(int m, int n) throws ArithmeticException, ArrayIndexOutOfBoundsException {

int div = m / n; // This may throw ArithmeticException if n is 0

String[] name2 = {"Mugilan", "java", "info"};

// Intentionally access an invalid index (for example, 3)

System.out.println(name2[3]); // This will throw ArrayIndexOutOfBoundsException

return div;

}

public static void main(String[] args) {

ThrowsExample obj = new ThrowsExample();

try {

System.out.println(divideNum(45, 9));

} catch (ArithmeticException e) {

System.out.println("\nNumber cannot be divided by 0");

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array Index is out of bound");

}

}

}

**Explanation:**

**🡪 Method Declaration**: The divideNum method declares that it can throw ArithmeticException and ArrayIndexOutOfBoundsException, indicating potential errors during division and array access.

**🡪 Exception Handling**: In the main method, we call divideNum(45, 9). Although this does not throw an ArithmeticException, it will attempt to access an invalid index of the array, leading to an ArrayIndexOutOfBoundsException.

**🡪Try-Catch Block**: The catch block catches the ArrayIndexOutOfBoundsException, and we print a message indicating that the array index is out of bounds, providing feedback on the error encountered.

**2 cases**

* **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
* **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

**Case 1: Handle Exception Using try-catch block**

* In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

**Case 2: Declare Exception**

* In case we declare the exception, if exception does not occur, the code will be executed fine.
* In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

**Note:**

* throws keyword will be activated at the runtime of the program.

**User-defined Exceptions**

* To create a user-defined exception, we need to perform the following steps.
* Create an exception class
* Implement user-defined exception

**Creating an Exception class**

public class AgeException extends RuntimeException{

public AgeException(){

System.out.println(“Invalid value for age”);

}

AgeException(String msg){

}

}

**Explanation:**

* The AgeException class is created in the above program.
* The AgeException class extends RuntimeException which is a subclass of the Exception class.
* The constructor of the AgeException() class prints the message to state that an invalid age value has been entered.
* In addition, it has an overloaded constructor which accepts string objects.

**Implementing User-defined Exception**

import java.util.\*;

public class ValidateAge{

public static void main(String[] args){

int age;

Scanner sc = new Scanner(System.in);

System.out.println(“Enter the age: ”);

age = sc.nextInt();

if(age<=0) {

try{

throw new AgeException();

}catch(AgeException e){

System.out.println(“Exception raised”);

}

} else{

System.out.println(“Age entered is: ”+age);

}

}

}

**Explanation:**

* In the above code, if the entered age is less than or equal to 0, the user-defined exception, AgeException is thrown.
* When the exception is thrown, the constructor defined in the class of the user-defined exception is invoked and the Invalid value for age message is displayed.

**Using the assert keyword**

* Assertions are used during the testing of a program.
* They enable you to test the assumption made in a program during its execution.

**Sample example Statement:**

* A particular program may require the value being passed to a method to be positive.
* You can test this by asserting that the value being passed to that particular method is greater than zero by using an assert keyword.

**Assertions benefits:**

* By using assertions, data can be validated easily.
* For example, in a method that performs division, you can use an assertion to validate at run-time if that method is performing division by zero.
* By using assertions, you can confirm whether the program is working as excepted.
* By using assertions, the task of debugging is simplified as assertions can easily indicate the source and the reason for an error.

**Implementing Assertions**

The ***assert*** keyword is used to implement assertions.

**Syntax:**

assert expressionA;

or

assert expression1 : expression2;

**Note:**

* Before you can successfully implement assertions in your code, you need to enable them in the java environment.
* In the above syntax, expressionA is Boolean expression.
* During the execution of program, expressionA in the assert statement is tested.
* If expressionA returns true, then:
* The assertion made in the program is true.
* The program execution continues uninterruptedly.
* No action takes place.
* However, if the expressionA returns false, then the assertion made in the program fails.
* The program throws an AssertionError object and the program execution terminates.

**Example:**

package com.demo.JavaCode;  
  
import java.util.Scanner;  
public class ValidateAge {  
 public static void main(String[] args){  
 int age; //Declaration variable  
 Scanner sc = new Scanner(System.*in*);  
 System.*out*.println("Enter the age: ");  
 age = sc.nextInt();  
 assert (age>0)&&(age<120);//statement  
 System.*out*.println("The entered age is: "+age);  
 }  
}

**Steps to Test the Program**

* **Compile the Program**: Make sure your program is compiled. In the terminal, navigate to your project directory and run:
* javac com/demo/JavaCode/ValidateAge.java
* **Run the Program with Assertions Enabled**: Run your program with the assertion flag enabled. In the terminal, use:
* java -ea com.demo.JavaCode.ValidateAge
* **Input Values**: When prompted to "Enter the age:", provide one of the following inputs:
* -1
* 0
* 120
* 150