Exception Handling

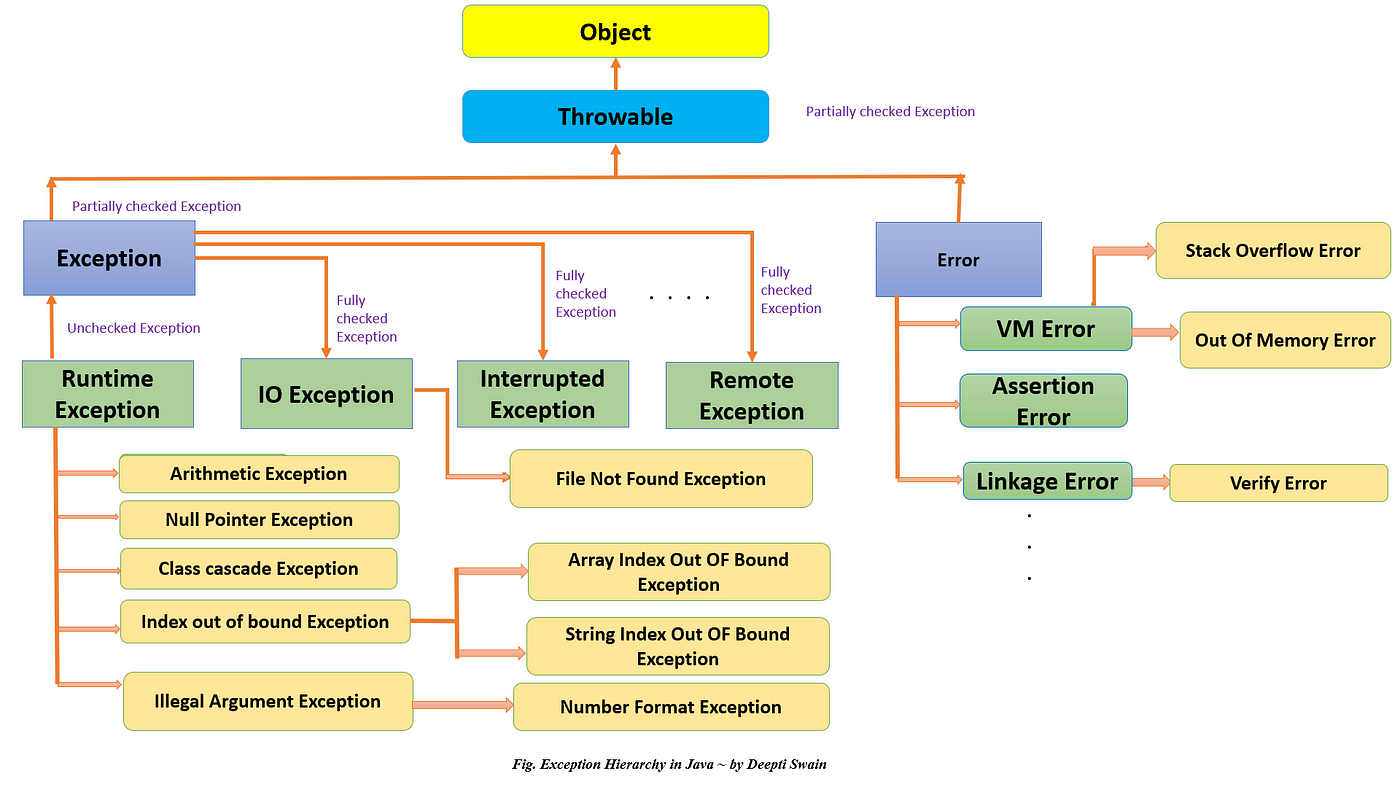
* Exception handling is a method used to manage and respond to exceptional conditions (i.e., errors or unexpected situations) that may occur during the execution of a program.
* These exceptional conditions can arise due to various reasons such as invalid user input, hardware failures, network issues, or unexpected behavior of the program itself.

Goals of Exception Handling:

1. **Graceful Termination:** Ensure that the program terminates gracefully without crashing when an unexpected situation occurs.
2. **Debugging and Logging**: Provide meaningful error messages and logs to aid in debugging and troubleshooting issues.
3. **Resource Cleanup**: Safely release resources (like files, database connections, network sockets, etc.) that were acquired during program execution.

Exception Dictionary Meaning: Exception is an abnormal condition.

# Hierarchy



**1. Throwable Class:**

* Throwable is the root class of the exception hierarchy in Java.
* It has two direct subclasses:-
  + Error
  + Exception

**2. Error Class:**

* Error represents serious issues that should typically not be caught or handled programmatically because they usually indicate irrecoverable problems in the JVM or system environment.

Examples include:

* OutOfMemoryError: Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory.
* StackOverflowError: Thrown when a stack overflow occurs because an application recurses too deeply.

**3. Exception Class:**

* Exception is a subclass of Throwable and serves as the superclass for all exceptions that are not errors.
* Exceptions are further categorized into two main types:
  + Checked Exceptions
  + Unchecked Exceptions (Runtime Exceptions)

**4. Checked Exceptions:**

* Checked Exceptions are exceptions that extend Exception but do not extend RuntimeException.
* They are checked by the compiler at compile-time, and you are required to either handle them using try-catch blocks or declare them in the method signature using throws clause.

Examples include:

* IOException: Signals that an I/O exception of some sort has occurred.
* SQLException: Indicates that an SQL exception has occurred during a database access operation.

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

* Unchecked Exceptions are exceptions that extend RuntimeException, which itself extends Exception.
* Unlike checked exceptions, they are not checked by the compiler at compile-time.
* Unchecked exceptions can occur at runtime and usually result from programming errors or exceptional conditions that are not expected to occur during normal execution.

Examples include:

* NullPointerException: Thrown when trying to access or call a method on a null object reference.
* ArithmeticException: Thrown when an arithmetic operation fails, such as division by zero.
* ArrayIndexOutOfBoundsException: Thrown when attempting to access an array element with an invalid index.

# Keywords-

* **try** The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
* **catch** The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
* **finally** The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
* **throw** The "throw" keyword is used to throw an exception.
* **throws** The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

1. **public** **class** JavaExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //code that may raise exception
5. **int** data=10/0;
6. }**catch**(ArithmeticException e){System.out.println(e);}
7. //rest code of the program
8. System.out.println("rest of the code...");
9. }
10. }

## Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

int a=50/0;//ArithmeticException

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

String s=null;

System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

String s="abc";

int i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

try Block:

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

**Syntax:**

try {

// Code that may throw exceptions

}

**Rules:**

* Must be followed by either a catch block or a finally block, or both.
* You can have multiple try blocks in a nested manner.

2. catch Block:

**Purpose**: The catch block is used to handle specific exceptions that may be thrown in the corresponding try block.

**Syntax:**

catch (ExceptionType e) {

// Exception handling code

}

**Rules:**

* Immediately follows the try block or a previous catch block.
* ExceptionType specifies the type of exception to catch. It should be a subclass of Throwable.
* You can have multiple catch blocks to handle different types of exceptions.

3. finally Block:

**Purpose:** The finally block contains code that needs to execute regardless of whether an exception was thrown or not.

**Syntax:**

finally {

// Code that always executes, regardless of exceptions

}

**Rules:**

* Optionally follows the try block and its associated catch block(s).
* Executes after the try block completes (either normally or due to an exception).
* Useful for resource cleanup (e.g., closing files, releasing database connections).

**4. Combination of try, catch, and finally**:

Syntax:

try {

// Code that may throw exceptions

} catch (ExceptionType1 e1) {

// Exception handling for ExceptionType1

} catch (ExceptionType2 e2) {

// Exception handling for ExceptionType2

} finally {

// Code that always executes, regardless of exceptions

}

Rules:

try block is mandatory.

catch block(s) are optional but must directly follow the try block.

finally block is optional but can only be used after a try block and its associated catch block(s).

**5. Exception Handling Best Practices:**

* Catch Specific Exceptions: Handle specific exceptions rather than catching Exception broadly.
* Use finally for Cleanup: Ensure resource cleanup (e.g., closing files, database connections) in the finally block.
* Avoid Swallowing Exceptions: Always include appropriate error logging or handling in catch blocks.
* Handle or Declare Checked Exceptions: Checked exceptions must either be handled using try-catch blocks or declared in the method signature using throws clause.





package data2;

public class \_1except1 {

public static void main(String[] args) {

int[] numbers = { 1, 2, 3, 4, 5 };

int index1 = 10;

int index2 = 0;

try {

// Outer try block

try

{

// Inner try block

int result = numbers[index1] / numbers[index2];

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

}

catch (ArithmeticException e)

{

System.out.println("Error: Division by zero");

}

// Simulating another nested try block in the same method

try {

String str = null;

System.out.println("Length of the string: " + str.length());

}

catch (NullPointerException e) {

System.out.println("Error: Null pointer exception");

}

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Error: Array index out of bounds");

}

}

}

//When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

Throw Keyword

* In Java, the throw keyword is used to explicitly throw an exception from a method or block of code.
* When an exception is thrown, the control is transferred to the nearest enclosing catch block that can handle that particular type of exception, or if no such block exists, the program terminates with an error message.

Throw use



1.

public class ThrowExample {

public static void main(String[] args) {

try {

validateAge(15);

} catch (IllegalArgumentException e) {

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

}

}

public static void validateAge(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age must be at least 18 years");

} else {

System.out.println("Valid age: " + age);

}

}

}

2 user defined exception.

* In Java, you can create your own custom exceptions by extending the Exception class (for checked exceptions) or the RuntimeException class (for unchecked exceptions).
* This allows you to define and throw exceptions that are specific to your application's requirements or domain logic

Custom-checked exception-

// Custom checked exception

class InvalidAgeException extends Exception {

// Constructor with a custom error message

public InvalidAgeException(String message) {

super(message);

}

}

// Example class using the custom exception

public class CustomExceptionExample {

// Method that validates age and throws the custom exception

public static void validateAge(int age) throws InvalidAgeException {

if (age < 0 || age > 150) {

throw new InvalidAgeException("Age must be between 0 and 150");

} else {

System.out.println("Valid age: " + age);

}

}

// Example usage

public static void main(String[] args) {

try {

validateAge(200); // Trying to validate an invalid age

} catch (InvalidAgeException e) {

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

}

}

}

**Runtime exception example**

// Custom runtime exception

class CustomRuntimeException extends RuntimeException {

// Constructor with a custom error message

public CustomRuntimeException(String message) {

super(message);

}

}

// Example class using the custom runtime exception

public class CustomRuntimeExceptionExample {

// Method that validates a positive number and throws the custom runtime exception

public static void validatePositiveNumber(int number) {

if (number < 0) {

throw new CustomRuntimeException("Negative number not allowed: " + number);

} else {

System.out.println("Valid number: " + number);

}

}

// Example usage

public static void main(String[] args) {

try {

validatePositiveNumber(-5); // Passing a negative number

} catch (CustomRuntimeException e) {

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

}

}

}

Throws

* In Java, the throws keyword is used in method signatures to declare that a particular method may throw one or more exceptions.
* It is part of the exception handling mechanism and is used to indicate to the caller of the method which exceptions might occur during its execution.

Syntax-

ReturnType methodName(ParameterList) throws ExceptionType1, ExceptionType2, ... {

// Method body

}

package data2;

public class \_5exc1 {

public static void main(String[] args) {

try{

validateAge(15); // Calling a method that may throw an exception

}

catch (Exception a){

System.out.println("this is exception.......");

}

System.out.println("Age validation passed.");

}

public static void validateAge(int age) throws Exception {

if (age < 18) {

System.out.println("Age must be 18 or older..................");

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

}

// Logic continues if age is valid

System.out.println("Age is valid: " + age);

}}