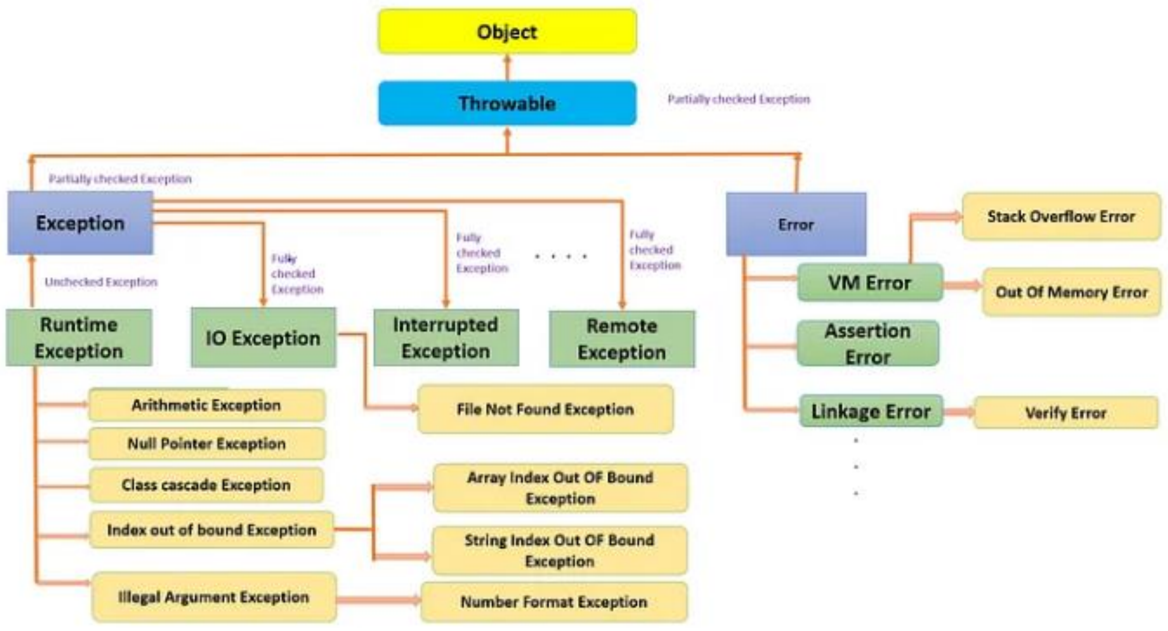
Exception Handling

Exception Hierarchy



Exception: An unwanted unexpected event that disturbs normal flow of the program is called exception.

Types of Exceptions:

1. Checked Exceptions:

   Checked exceptions are those that the compiler forces you to handle explicitly.

   Examples include IOException, SQLException, etc.

   Handling checked exceptions is done using the try-catch block.

2. Unchecked Exceptions:

   Unchecked exceptions are not checked at compile-time but are thrown at runtime.

   Examples include NullPointerException, ArrayIndexOutOfBoundsException, etc.

   Handling unchecked exceptions is optional but recommended.

Note: RuntimeException and its child classes, Error and its child classes are unchecked and all the remaining are considered as checked exceptions.

Note: Whether exception is checked or unchecked compulsory it should occurs at

runtime only and there is no chance of occurring any exception at compile time

Fully checked Vs Partially checked:

A checked exception is said to be fully checked if and only if all its child classes are also checked.

Example:

1) IOException

2) InterruptedException

A checked exception is said to be partially checked if and only if some of its child classes are unchecked.

Example:

Exception

Note: The only possible partially checked exceptions in java are:

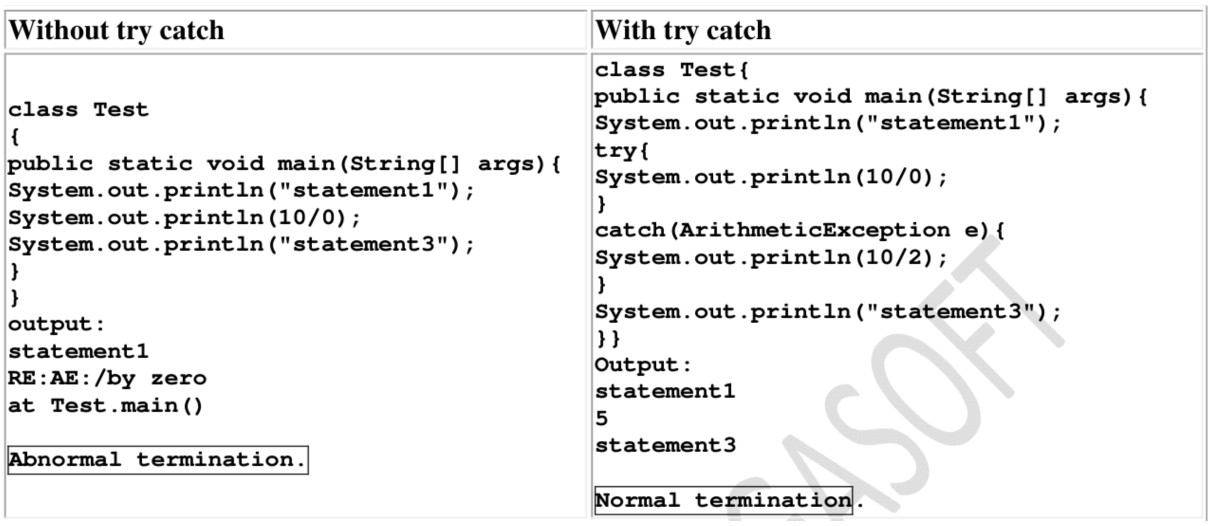
1. Throwable.

2. Exception.

Customized Exception Handling by using try-catch:

1. It is highly recommended to handle exceptions.
2. In our program the code which may raise exception is called risky code, we have to place risky code inside try block and the corresponding handling code inside catch block.

```
try {
        //risky code here
} catch (Exception ex) {
   // Handle specific exceptio
}
```



Without try catch

```
class Test
{
public static void main(String[] args){
System.out.println("statement1");
System.out.println(10/0);
System.out.println("statement3");
}
}
output:
statement1
RE:AE:/by zero
at Test.main()

Abnormal termination.
```

With try catch

```
class Test{
public static void main(String[] args){
System.out.println("statement1");
try{
System.out.println(10/0);
}
catch(ArithmeticException e){
System.out.println(10/2);
}
System.out.println("statement3");
}}
Output:
statement1
5
statement3

Normal termination.
```

Note:

1. Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception. Hence, we have to place/take only risk code inside try block and length of the try block should be as less as possible.

2. If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.

3. There may be a chance of raising an exception inside catch and finally blocks also in addition to try block.

Various methods to print exception information:

1. printStackTrace():

    This method prints exception information in the following format.Name of the exception: description of exception Stack trace

2. toString():

    This method prints exception information in the following format. Name of the exception: description of exception

3. getMessage():

    This method returns only description of the exception.Description

Try with multiple catch blocks

The way of handling an exception is varied from exception to exception. Hence for every exception type it is recommended to take a separate catch block. That is try with multiple catch blocks is possible and recommended to use.

EXAMPLE:

```
try {
      int result = numbers[0] / numbers[1];
      return result;
    } catch (ArithmeticException ex) {
      // Handle arithmetic exception
      throw new ArithmeticException("Division by zero not allowed");
    } catch (ArrayIndexOutOfBoundsException ex) {
      // Handle array index out of bounds exception
      throw new ArrayIndexOutOfBoundsException("Array index out of bounds");
    }
```

Finally block:

1. It is not recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.
2. It is not recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.
3. We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled. Such type of best place is nothing but finally block.
4. Hence the main objective of finally block is to maintain cleanup code.

```
try {
      // Code that may throw an exception
      int result = divide(10, 0);
      System.out.println("Result: " + result);
    } catch (ArithmeticException ex) {
      // Handle specific exception
      System.err.println("ArithmeticException: " + ex.getMessage());
} finally {
    // Cleanup code or resource release
    System.out.println("Finally block executed");
}
```

Case-1: If there is no Exception:

```
try
{
    System.out.println("try block executed");
}
catch(ArithmeticException e)
{
    System.out.println("catch block executed");
}
finally
{
    System.out.println("finally block executed");
}
Output:
try block executed
Finally block executed
```

Case-2: If an exception raised but the corresponding catch block matched:

```
try
{
    System.out.println("try block executed");
    System.out.println(10/0);
}
catch(ArithmeticException e)
{
    System.out.println("catch block executed");
}
finally
{
    System.out.println("finally block executed");
}

Output:
Try block executed
Catch block executed
Finally block executed
```

return Vs finally:

Even though return statement presents in try or catch blocks first finally will be executed and after that only return statement will be considered. finally block dominates return statement.

```
try
{
    System.out.println("try block executed");
    return;
}
catch(ArithmeticException e)
{
    System.out.println("catch block executed");
}
finally
{
    System.out.println("finally block executed");
}
Output:
try block executed
Finally block executed
```

If return statement present try, catch and finally blocks then finally block return statement will be considered.

Example:

```
  try
        {
            System.out.println(10/0);
            return 777;
        }
        catch(ArithmeticException e)
        {
            return 888;
        }
        finally{
            return 999;
        }
    }
Output:
        999
```

Note:
finally vs System.exit(0):

There is only one situation where the finally block won't be executed is whenever we are using System.exit(0) method. Whenever we are using System.exit(0) then JVM itself will be shutdown , in this case finally block won't be executed.

i.e., System.exit(0) dominates finally block.

Difference between final, finally, and finalize:

final:

1. final is the modifier applicable for classes, methods and variables.
2. If a class declared as the final then child class creation is not possible.
3. If a method declared as the final then overriding of that method is not possible.
4. If a variable declared as the final then reassignment is not possible.

finally:

finally is the block always associated with try-catch to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled.

finalize:

finalize is a method, always invoked by Garbage Collector just before destroying an object to perform cleanup activities.

throw statement:

In Java, the throws clause is used in a method signature to indicate that the method might throw certain types of exceptions during its execution. When a method is declared with a throws clause, it means that the responsibility of handling those exceptions is passed to the caller of the method.

```
public static void readFile(String filename) throws
FileNotFoundException, IOException {
    FileReader fileReader = null;
    try {
        fileReader = new FileReader(filename);
        // Code to read from the file
    } finally {
        if (fileReader != null) {
            fileReader.close();
        }
    }
}
```

```
public static int readIntegerFromConsole() throws
InputMismatchException {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    try {
        return scanner.nextInt();
    } finally {
        scanner.close();
    }
}
```

throw statement:

he throw statement in Java is used to explicitly throw an exception. It is typically used in methods to indicate that a specific exceptional condition has occurred, and the normal flow of the program should be interrupted.

```
public static int divideNumbers(int dividend, int divisor) {
    if (divisor == 0) {
        // Using throw to explicitly throw an exception
        throw new ArithmeticException("Cannot divide by
zero");
    }
    return dividend / divisor;
}
```