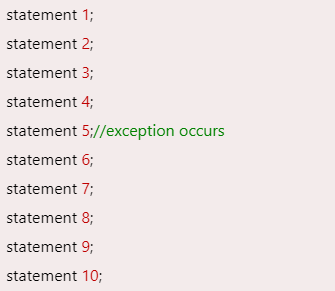
**Chapter 1 – Unit 3**

**Exception Handling**

Concepts of Exception handling, types of exceptions, usage of try, catch, throw, throws and finally keywords, multiple catch clauses, nested try, Built-in exceptions, creating own exception sub classes.

**Introduction of Exception handling**

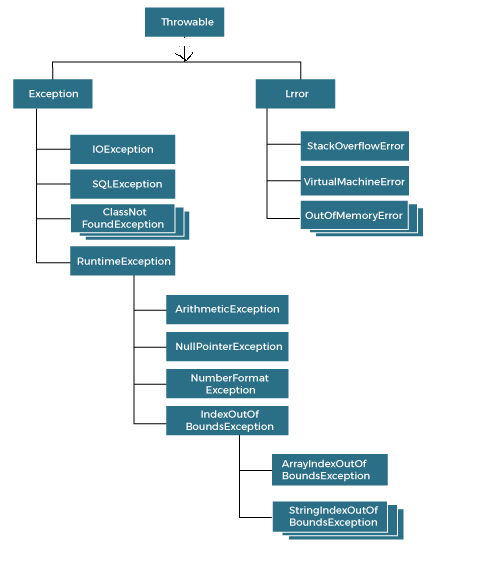
* In Java, **an exception** is an event that disrupts (disturbance or problem) the normal flow of the program. It is an object which is thrown at runtime.
* **Exception Handling** is a mechanism to handle runtime errors.
* **Advantage of Exception Handling**
* **The core advantage of exception handling is to maintain the normal flow of the application.**
* An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.
* Let's consider a scenario:

****

* Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

**Hierarchy of Java Exception classes**

* The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



**Types of Java Exceptions**

* There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:
* Checked Exception
* Unchecked Exception
* Error



* **1) Checked Exception**
* **Checked exceptions are checked at compile-time.**
* The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions.
* For example, IOException, SQLException, etc.
* **2) Unchecked Exception**
* **Unchecked exceptions are not checked at compile-time, but they are checked at runtime.**
* The classes that inherit the RuntimeException are known as unchecked exceptions.
* For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.
* **3) Error**
* **Error is irrecoverable.**
* Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

**Exception Keywords**

* Java provides five keywords that are used to handle the exception. The following table describes each.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| 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. |

Exception Handling Example

* Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

classExceptionExample

{

public static void main(String args[])

{

try

{

int data=100/0;

}

catch(ArithmeticException e)

{

System.out.println(e);

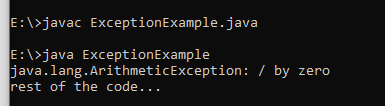
}

System.out.println("rest of the code...");

}

}

OUTPUT :



**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](https://www.javatpoint.com/java-variables), 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](https://www.javatpoint.com/java-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.

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

**try-catch block**

**try block**

* **try block** is used to enclose the code that might throw an exception. It must be used within the method.
* If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
* try block must be followed by either catch or finally block.

**Syntax of try-catch**

try

{

//code that may throw an exception

}

catch(Exception\_class\_Name  ref)

{

}

**Syntax of try-finally block**

try

{

//code that may throw an exception

}

finally

{

}

**catch block**

* Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
* The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of try-catch block



* The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:
* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.
* But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Example :without exception handling

class TryCatchExample1

{

public static void main(String[] args)

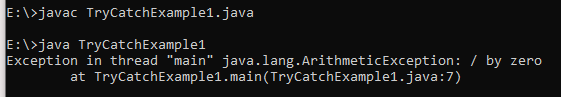
{

int data=50/0; //may throw exception

System.out.println("rest of the code");

}

}



* As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).
* There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

* Let's see the solution of the above problem by a java try-catch block.

class TryCatchExample2

{

public static void main(String[] args)

{

try

{

int data=50/0; //may throw exception

}

//handling the exception

catch(ArithmeticException e)

{

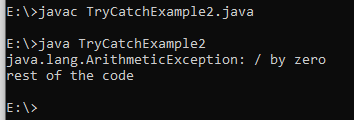
System.out.println(e);

}

System.out.println("rest of the code");

}

}



* As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 3

* Let's see an example to handle another unchecked exception.
* TryCatchExample3.java

class TryCatchExample3

{

public static void main(String[] args)

{

try

{

intarr[]= {1,3,5,7};

System.out.println(arr[10]); //may throw exception

}

// handling the array exception

catch(ArrayIndexOutOfBoundsException e)

{

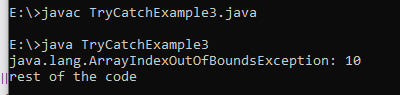
System.out.println(e);

}

System.out.println("rest of the code");

}

}



**Catch Multiple Exceptions**

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.

Flowchart of Multi-catch Block



Example 1

Let's see a simple example of java multi-catch block.

**MultipleCatchBlock1.java**

class MultipleCatchBlock1

{

public static void main(String[] args)

{

try

{

int a[]=new int[5];

a[5]=30/0;

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception occurs");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("ArrayIndexOutOfBounds Exception occurs");

}

catch(Exception e)

{

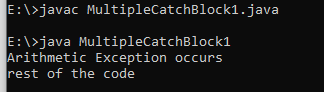
System.out.println("Parent Exception occurs");

}

System.out.println("rest of the code");

}

}



Example 2

Let's see a simple example of java multi-catch block.

**MultipleCatchBlock2.java**

class MultipleCatchBlock2

{

public static void main(String[] args)

{

try

{

int a[]=new int[5];

System.out.println(a[10]);

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception occurs");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("ArrayIndexOutOfBounds Exception occurs");

}

catch(Exception e)

{

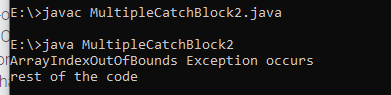
System.out.println("Parent Exception occurs");

}

System.out.println("rest of the code");

}

}



Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

**MultipleCatchBlock3.java**

class MultipleCatchBlock3

{

public static void main(String[] args)

{

try

{

int a[]=new int[5];

a[5]=30/0;

System.out.println(a[10]);

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception occurs");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("ArrayIndexOutOfBounds Exception occurs");

}

catch(Exception e)

{

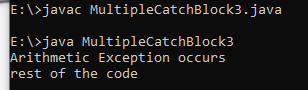
System.out.println("Parent Exception occurs");

}

System.out.println("rest of the code");

}

}



### Example 4

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

**MultipleCatchBlock4.java**

class MultipleCatchBlock4

{

public static void main(String[] args)

{

try

{

String s=null;

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

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception occurs");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("ArrayIndexOutOfBounds Exception occurs");

}

catch(Exception e)

{

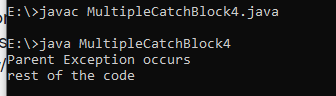
System.out.println("Parent Exception occurs");

}

System.out.println("rest of the code");

}

}



**Nested try block**

* In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.
* For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

### Why use nested try block

* Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### Syntax:



Example 1

Let's see an example where we place a try block within another try block for two different exceptions.

class NestedTryBlock

{

public static void main(String args[])

{

//outer try block

try

{

//inner try block 1

try

{

System.out.println("going to divide by 0");

int b =39/0;

}

//catch block of inner try block 1

catch(ArithmeticException e)

{

System.out.println(e);

}

//inner try block 2

try

{

int a[]=new int[5];

//assigning the value out of array bounds

a[5]=4;

}

//catch block of inner try block 2

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println(e);

}

System.out.println("other statement");

}

//catch block of outer try block

catch(Exception e)

{

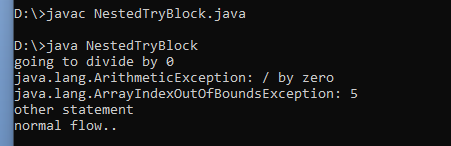
System.out.println("handled the exception (outer catch)");

}

System.out.println("normal flow..");

}

}



**Note :**

* 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.
* If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

Example 2

* Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.
* NestedTryBlock2.java

class NestedTryBlock2

{

public static void main(String args[])

{

// outer (main) try block

try

{

//inner try block 1

try

{

// inner try block 2

try

{

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

//printing the array element out of its bounds

System.out.println(arr[10]);

}

// to handles ArithmeticException

catch (ArithmeticException e)

{

System.out.println("Arithmetic exception");

System.out.println(" inner try block 2");

}

}

// to handle ArithmeticException

catch (ArithmeticException e)

{

System.out.println("Arithmetic exception");

System.out.println("inner try block 1");

}

}

// to handle ArrayIndexOutOfBoundsException

catch (ArrayIndexOutOfBoundsException e4)

{

System.out.print(e4);

System.out.println(" outer (main) try block");

}

catch (Exception e5)

{

System.out.print("Exception");

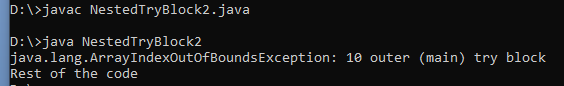
System.out.println(" handled in main try-block");

}

System.out.print("Rest of the code");

}

}



**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.

### Flowchart of finally 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.

## **Usage of Java finally**

* Let's see the different cases where Java finally block can be used.

Case 1: When an exception does not occur

* Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

**TestFinallyBlock.java**

class TestFinallyBlock

{

public static void main(String args[])

{

Try

{

//below code do not throw any exception

int data=25/5;

System.out.println(data);

}

//catch won't be executed

catch(NullPointerException e)

{

System.out.println(e);

}

//executed regardless of exception occurred or not

finally

{

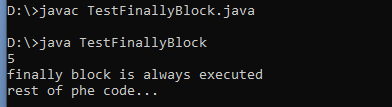
System.out.println("finally block is always executed");

}

System.out.println("rest of phe code...");

}

}



Case 2: When an exception occurr but not handled by the catch block

* Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlocclass TestFinallyBlock1

{

public static void main(String args[])

{

try

{

System.out.println("Inside the try block");

//below code throws divide by zero exception

int data=25/0;

System.out.println(data);

}

//cannot handle Arithmetic type exception

//can only accept Null Pointer type exception

catch(NullPointerException e)

{

System.out.println(e);

}

//executes regardless of exception occured or not

finally

{

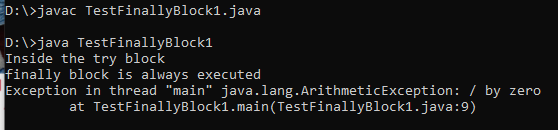
System.out.println("finally block is always executed");

}

System.out.println("rest of the code...");

}

}



Case 3: When an exception occurs and is handled by the catch block

**Example:**

* Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

public static void main(String args[])

{

try

{

System.out.println("Inside try block");

//below code throws divide by zero exception

int data=25/0;

System.out.println(data);

}

//handles the Arithmetic Exception / Divide by zero exception

catch(ArithmeticException e)

{

System.out.println("Exception handled");

System.out.println(e);

}

//executes regardless of exception occured or not

finally

{

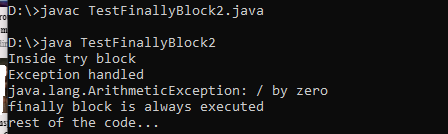
System.out.println("finally block is always executed");

}

System.out.println("rest of the code...");

}

}



**throw keyword**

* **The Java throw keyword is used to throw an exception explicitly.**
* We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
* We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.
* We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.
* **The syntax of the Java throw keyword is given below.**
* **throw Instance i.e.,**

****

* Let's see the example of throw IOException.



Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

### Example 1: Throwing Unchecked Exception

* In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

**TestThrow1.java**

* In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

class TestThrow1

{

//function to check if person is eligible to vote or not

public static void validate(int age)

{

if(age<18)

{

//throw Arithmetic exception if not eligible to vote

throw new ArithmeticException("Person is not eligible to vote");

}

else

{

System.out.println("Person is eligible to vote!!");

}

}

//main method

public static void main(String args[])

{

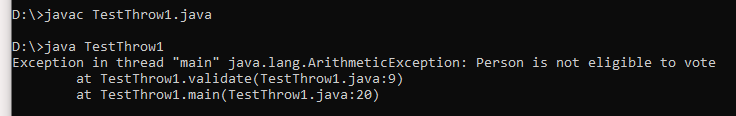
//calling the function

validate(13);

System.out.println("rest of the code...");

}

}



**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.

### Syntax of Java throws

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

{

//method code

}

Advantage of Java throws keyword

* Now Checked Exception can be propagated (forwarded in call stack).
* It provides information to the caller of the method about the exception.

## **throws Example**

* Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

import java.io.IOException;

class Testthrows1

{

void m()throws IOException

{

throw new IOException("device error");//checked exception

}

void n()throws IOException

{

m();

}

void p()

{

try{

n();

}

catch(Exception e)

{

System.out.println("exception handled");

}

}

public static void main(String args[])

{

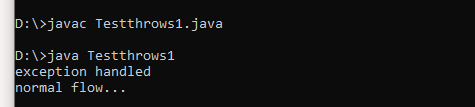
Testthrows1 obj=new Testthrows1();

obj.p();

System.out.println("normal flow...");

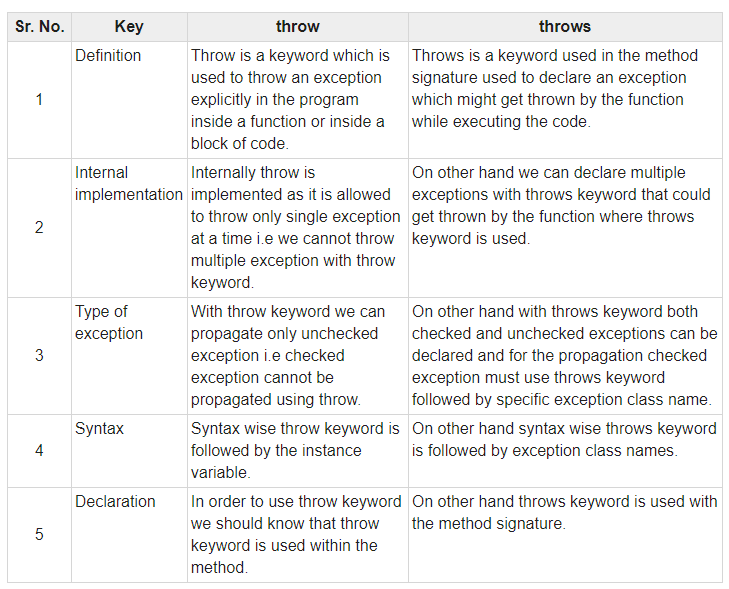
}

}



**Difference between throw and throws in Java**

* The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.
* There are many differences between [throw](https://www.javatpoint.com/throw-keyword) and [throws](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws) keywords. A list of differences between throw and throws are given below:



## **throw Example**

**TestThrow.java**

class TestThrow

{

//defining a method

public static void checkNum(int num)

{

if (num < 1)

{

throw new ArithmeticException("\nNumber is negative, cannot calculate square");

}

else

{

System.out.println("Square of " + num + " is " + (num\*num));

}

}

//main method

public static void main(String[] args) {

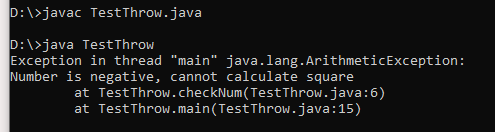
TestThrow obj = new TestThrow();

obj.checkNum(-3);

System.out.println("Rest of the code..");

}

}



## **throws Example 1:**

**TestThrows.java**

class TestThrows

{

//defining a method

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

{

int div = m / n;

return div;

}

//main method

public static void main(String[] args)

{

TestThrows obj = new TestThrows();

try

{

System.out.println(obj.divideNum(45, 0));

}

catch (ArithmeticException e)

{

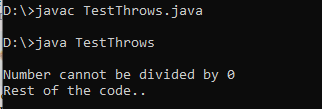
System.out.println("\nNumber cannot be divided by 0");

}

System.out.println("Rest of the code..");

}

}



## **Example 2:**

**TestThrows.java**

class TestThrows

{

//defining a method

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

{

int div = m / n;

return div;

}

//main method

public static void main(String[] args)

{

TestThrows obj = new TestThrows();

try

{

System.out.println(obj.divideNum(45, 1));

}

catch (ArithmeticException e)

{

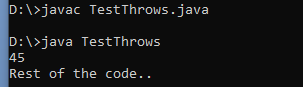
System.out.println("\nNumber cannot be divided by 0");

}

System.out.println("Rest of the code..");

}

}



## **throw and throws Example**

**TestThrowAndThrows.java**

**class TestThrowAndThrows**

**{**

**// defining a user-defined method**

**// which throws ArithmeticException**

**static void method() throws ArithmeticException**

**{**

**System.out.println("Inside the method()");**

**throw new ArithmeticException("throwing ArithmeticException");**

**}**

**//main method**

**public static void main(String args[])**

**{**

**try**

**{**

**method();**

**}**

**catch(ArithmeticException e)**

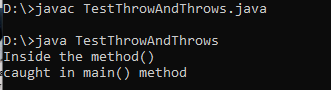
**{**

**System.out.println("caught in main() method");**

**}**

**}**

**}**



**Final – finally – Finalize()**

## Final

* It is a keyword.
* It is used to apply restrictions on classes, methods and variables.
* It can’t be inherited.
* It can’t be overridden.
* Final methods can’t be inherited by any class.
* It is needed to initialize the final variable when it is being declared.
* Its value, once declared, can’t be changed or re-initialized.

## Finally

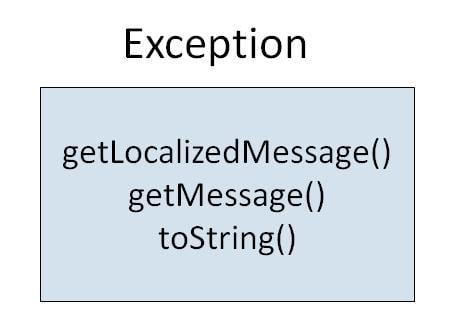
* It is a block.
* It is used to place important code in this block.
* It gets executed irrespective of whether the exception is handled or not.

## Finalize

* It is a method.
* It is used to perform clean up processing right before the object is collected by garbage collector.

**Custom Exception or User defined Exceptions**

* **In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need. (OR)**
* **Java user-defined exception is a custom exception created and throws that exception using a keyword ‘throw’. It is done by extending a class ‘Exception’. (OR)**
* **Java allows the user to create its own user-defined exception or custom exception simply by declaring a subclass of exception class and using throw keyword.**
* **There are some rules also to create Exception classes.**
* **Constructor:** This is not mandatory in creating any constructor in the custom exception class. Providing parameterized constructors in the custom exception class is a good practice.
* **Naming Convention:** All exception classes are provided by the JDK end; hence, a custom exception should follow a naming convention.
* **Extends Exception class:** If the user is creating a custom exception class, then the user has to extend the Exception class.
* **The available methods in** exception class **are :**



* There is no need to override any of the above methods available in the Exception class, in your derived class. But practically, you will require some amount of customizing as per your programming needs.
* **Syntax:**
* We do not have any particular syntax for Java user-defined exception;
* we will see how to create a User-defined exception.

**class SampleException**

**{  
public static void main(String args[])**

**{  
try**

**{  
throw new UserException(<value>); // used to create new exception and throw  
}  
catch(Exception e)**

**{  
System.out.println(e);  
}  
}  
}  
class UserException extends Exception**

**{  
// code for for exception class  
}**

## **Example 1 :**

**class OwnException extends Exception**

**{**

**// Declare default constructor.**

**OwnException()**

**{**

**}**

**}**

**class MyClass**

**{**

**public static void main(String[] args)**

**{**

**try**

**{**

**// Create an object of user defined exception and throw it using throw clause.**

**OwnException obj = new OwnException();**

**throw obj;**

**}**

**catch (OwnException ex)**

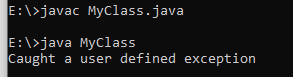
**{**

**System.out.println("Caught a user defined exception");**

**}**

**}**

**}**

****

**Note :**

* The **getMessage()** method of Throwable class is used to return a detailed message of the Throwable object which can also be null.
* One can use this method to get the detail message of exception as a string value.
* Return Value: This method returns the detailed message of this Throwable instance. // the getMessage() Method.
* The **getLocalizedMessage()** method of Throwable class is used to get a locale-specific description of the Throwable object when an Exception Occurred.
* For the subclasses which do not override this method, the default implementation of this method returns the same result as getMessage().

## **Example 2 :**

**class OwnException extends Exception**

**{**

**// Declare parameterized constructor with String as a parameter.**

**OwnException(String str)**

**{**

**super(str); // Call super exception class constructor.**

**}**

**}**

**class MyClass**

**{**

**public static void main(String[] args)**

**{**

**try**

**{**

**// Create an object of user defined exception and throw it using throw clause.**

**OwnException obj = new OwnException("Creating user defined exception");**

**throw obj;**

**// or, throw new OwnException("Creating user defined exception");**

**}**

**catch (OwnException ex)**

**{**

**System.out.println("Caught a user defined exception");**

**System.out.println(ex.getMessage());**

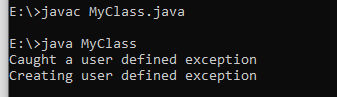
**//System.out.println(ex.getLocalizedMessage());**

**//System.out.println(ex.toString());**

**}**

**}**

**}**

****

## **Example 3:**

Let’s take an example program where we will evaluate candidate’s age to vote. If the candidate’s age is less than 18 years, the program will throw a custom exception “Invalid age”. See the program source code to understand better and follow all steps.

import java.util.Scanner;

class InvalidAgeException extends Exception

{

// Declare a parameterized exception with string str as a parameter.

InvalidAgeException(String str)

{

super(str);

}

}

class MyClass

{

private static int age;

static void validate() throws InvalidAgeException

{

Scanner sc = new Scanner(System.in);

System.out.println("Enter your age");

age = sc.nextInt();

if(age < 18)

throw new InvalidAgeException("Invalid Age, You are not eligible to vote");

else

System.out.println("Welcome to vote");

}

public static void main(String[] args)

{

try

{

validate();

}

catch(Exception e)

{

System.out.println("Caught an Exception: \n "+e);

System.out.println(e.getMessage());

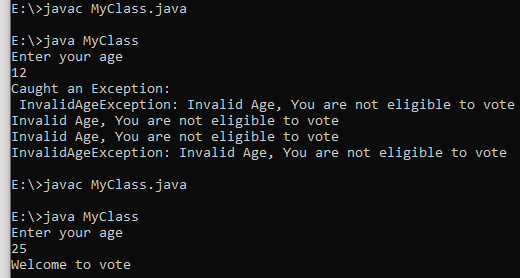
System.out.println(e.getLocalizedMessage());

System.out.println(e.toString());

}

}

}

****

**toString() in Java**

* So what exactly is this method? Object class is the parent class in Java.
* It contains the toString() method.
* The toString() method is used to return a string representation of an object.
* If any object is printed, the toString() method is internally invoked by the java compiler.
* Else, the user implemented or overridden toString() method is called.

Here are some of the advantages of using this method.

**Advantage**

If you override the toString() method of the Object class, it will return values of the object, hence you are not required to write a lot of code.

**Example For toString**

|  |  |
| --- | --- |
|  | public class Employee  {  int id;  String name;  String city;  Employee(int id, String name, String city)  {  this.id=id;  this.name=name;  this.city=city;  }  public static void main(String args[]){  Employee e1=new Employee(01,"Ari","NewYork");  Employee e2=new Employee(02,"Jon","Chicago");  System.out.println(e1);//compiler writes here e1.toString()  System.out.println(e2);//compiler writes here e2.toString()  }  } |

**Output:**

Employee@6d06d69c

Employee@7852e922

The code prints the HashCode values of the objects in the example.

**Throwable Class toString() method**

* **toString() Method** is available in java.lang package.
* **toString() Method** is used to return a short description of the exception.
* **toString() Method** is a non-static method, it is accessible with the class object only and if we try to access the method with the class name then we will get an error.
* **toString() Method** does not throw an exception at the time of string representation about the exception.

**Syntax:**

public String toString();

**Parameter(s):**

* It does not accept any parameter.

**Return value:**

The return type of the method is String, it represents string for the exception.

**// Java program to demonstrate the example of String toString() method of Throwable**

public class ToString

{

public static void main(String args[]) throws Exception

{

try

{

//div(1,1);

div(-3, 0);

}

catch (Exception ex)

{

System.out.println("ex.toString() :" + ex.toString());

}

}

// This method divide number by 0

public static void div(int d1, int d2) throws Exception

{

int res = d1 / d2;

System.out.println("res :" + res);

}

}

**Output 1:**

1

**Output 2:**

ex.toString() :java.lang.ArithmeticException: / by zero