## Exception Handling

An *exception* is an abnormal condition that arises in a code sequence at run time. A Java exception is an object that describes an exceptional condition that has occurred in a piece of code. Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Program statements that should be monitored for exceptions are contained within a **try** block. **catch** block is used to handle the exceptions. To manually throw an exception, the keyword **throw** is used . Any exception that is thrown out of a method must be specified by a **throws** clause. Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

The general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

All exception types are subclasses of the built-in class **Throwable**. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class should be sub classed to create user defined, custom exception types. **RuntimeException** is an important subclass of Exception. It includes things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by the program. Exceptions of type **Error** are used by the Java run-time system to indicate errors with respect to the run-time environment, itself. Stack overflow is an example of such an error.

**Common Runtime Exceptions**

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NullPointerException
- NumberFormatException

**ArithmeticException:**

The arithmetic exception base class is java.lang.ArithmeticException, which is the child class of java.lang.RuntimeException, which in turn is the child class of java.lang.Exception. An ArithmeticException can occur when we perform a division where 0 is used as a divisor.

```
class Test{
public static void main(String args[]) {
        Scanner s = new Scanner(System.in);
        int a = s.nextInt();
        int b = 10/a;
        System.out.println("Code");
}
```

In the program, if the user provides input as "0", then an ArithmeticException occurs.

**ArrayIndexOutOfBoundsException:**

The ArrayIndexOutOfBoundsException occurs whenever we are trying to access any item of an array at an index which is not present in the array. In other words, the index may be negative or exceeded the size of an array.

```
public class ArrayIndexOutOfBoundsException {

   public static void main(String[] args) {
String[] arr = {"Rohit","Shikar","Virat","Dhoni"};

      for(int i=0;i<=arr.length;i++) {
         System.out.println(arr[i]);
      }
   }
}
```

In the example, the for loop tries to access index 4 which is unavailable thus giving rise to ArrayIndexOutOfBoundsException.

**NullPointerException:**

Null Pointer Exception is a kind of run time exception that is thrown when the java program attempts to use the object reference that contains the null value.

```
public class NullPointer {
        static void length(String str) {
                int len = str.length();
                System.out.print(len);
        }
        public static void main(String ar[]) {
                String s = null;
```

```
                length(s);
        }

}
```

In the program above, String s is pointing to a “null”. Using length on this null value causes a NullPointerException.

**NumberFormatException**

The NumberFormatException is an exception in Java that occurs when an attempt is made to convert a string with an incorrect format to a numeric value. This exception is thrown when it is not possible to convert a string to a numeric type.

```
public class NumberFormat {

        static void proc(String str) {
                Integer.parseInt(str);
        }

        public static void main(String ar[]) {
                String s = "Hello";
                proc(s);
        }
}
```

In this example the string “hello” cannot be converted to an Integer using parseInt method. Hence NumberFormatException is thrown.

**Uncaught Exceptions**

Consider the code below:

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
        }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop. Since an exception handlers of our own is not written, the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by the program will be processed by the default handler and causes termination of the program. The default handler displays a string describing the exception and prints a stack trace from the

point at which the exception occurred. When the above code is executed the following is displayed.

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

The stack trace shown above indicates that :

In the class named **Exc0**, in the method **main**, within the file named **Exc0.java** an error has occurred at **line 4**. Also the type of exception is **ArithmeticException.**

**Using try and catch**

Although the default exception handler provided by the Java run-time system is useful for debugging, it is better if the user handles the exception, as it prevents the program from automatically terminating.

The code that has to be monitored is enclosed inside a **try** block. Following the **try** block, a **catch** block should be written. This block must specify the exception type that is to be caught. The program below includes a **try** block and a **catch** clause that processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
	public static void main(String args[])
	{
		int d, a;
		try { // monitor a block of code.
		d = 0;
		a = 42 / d;
		}

		catch (ArithmeticException e) { // catch divide-by-zero error
		System.out.println("Division by zero.");
		}
		System.out.println("After catch statement.");
	}
}
```

The program generates the following output.

```
Division by zero.
After catch statement.
```

The program executes normally after the catch block and does not abruptly terminate unlike the previous example. This is indeed useful. If there are hundreds of lines of code past the erroneous statement, they will be executed normally after the exception is handled. If not handled, hundred lines of code would have remained unexecuted.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. The goal of most well-

constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

**Multiple catch clauses**

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, two or more **catch** clauses can be specified, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block.

```
class MultipleCatches {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[] = { 1 };
                        c[42] = 99;
                   }
                catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                        }
                catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index oob: " + e);
                        }

        System.out.println("After try/catch blocks.");
        }
}
```

This program will cause a division-by-zero exception if it no command line arguments are given. If a command line argument is given, then the value of **a** will be greater than zero and hence no division-by-zero exception occurs. But it will cause an **ArrayIndexOutOfBoundsException**, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

Output:

When no command line argument is given:

```
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

When 1 command line argument is passed:

```
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

**Nested try statements**

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack.

If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds. If no **catch** statement matches, then the Java run-time system will handle the exception.

```
class NestTry {
		public static void main(String args[]) {
			try {
				int a = args.length;
				int b = 42 / a;
				System.out.println("a = " + a);

				try {

				if(a==1) a = a/(a-a);
				if(a==2) {
					int c[] = { 1 };
					c[42] = 99;
				}
			  }

			catch(ArrayIndexOutOfBoundsException e) {
				System.out.println("Array index out-of-bounds: " + e);
				}
			}
			catch(ArithmeticException e) {
			System.out.println("Divide by 0: " + e);
			}
	}
}
```

The program nests one try within another. When the program is executed with no command line arguments, a divide by zero exception is generated by the outer try block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If the program is executed with two command-line arguments an array boundary exception is generated from within the inner **try** block.

throw

To throw an exception explicitly, a **throw** statement is used. The general form of **throw** is

throw *ThrowableInstance*;

*ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. There are two ways to obtain a **Throwable** object:

- creating one with the **new** operator.
- using a parameter in a **catch** clause

The flow of execution stops immediately after the **throw** statement. Any subsequent statements are not executed.

Example 1: (Creating with new operator)

```
public class MyClass {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    } else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15);
System.out.println(“Rest of the code”);
  }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.
        at MyClass.checkAge(MyClass.java:4)
 at MyClass.main(MyClass.java:12)
```

Example 2: (Using parameter of catch)

```
public class Test{
   static void test()
   {
      try{
         int res= 4/0;
      }
      catch(ArithmeticException e)
      {
         System.out.println("Caught in test");
         throw e;
      }
   }

   public static void main(String args[]){
      try
      {
         test();
      }
      catch(ArithmeticException e)
      {
         System.out.println("Caught in main");
      }
      System.out.println("Rest of the code");
   }
}
```

Output

Caught in test

Caught in main

Rest of the code

**throws**

If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. This is done by including a **throws** clause in the method's declaration.

This is the general form of a method declaration that includes a **throws** clause:

*type method-name*(*parameter-list*) throws *exception-list*

{

// body of method

}

*exception-list* is a comma-separated list of the exceptions that a method can throw.

```
public class TestThrow1{

  static void validate(int age) throws ArithmeticException {
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }

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

   validate(13);
  }
  catch(ArithmeticException e)
{
   System.out.println("Caught");
}

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

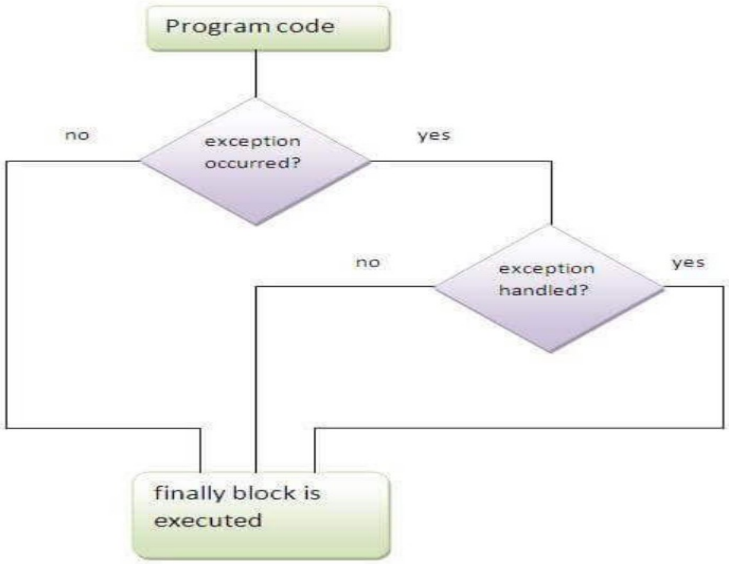
Output

```
Caught
rest of the code...
```

In the above code the method **validate** explicitly specifies that it will throw a Arithmetic Exception and hence the caller of that method should take care to provide necessary exception handling mechanism.

## finally

Java finally block is a block that is used to execute important code such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.




Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

**Case 1: No Exception occurs**

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
Output:
        5
     finally block is always executed
     rest of the code...
```

**Case 2 : Exception occurs but is not handled**

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
Output:

finally block is always executed
       Exception in thread main java.lang.ArithmeticException:/ by zero
```

**Case 3: Exception occurs and is handled**

```
public class TestFinallyBlock2{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

```
Output:
Exception in thread main java.lang.ArithmeticException:/ by zero
       finally block is always executed
       rest of the code...
```

In all the above three case the finally block will be executed

## User Defined Exceptions

Although Java's built-in exceptions handle most errors, there could be a need to create Exceptions of our own to handle situations specific to our applications. This can be done by defining a subclass of **Exception.**

The Exception class does not have any methods of its own. It inherits those methods provided by Throwable. All Exceptions including the ones created by the user have methods defined by Throwable available to them. Exceptions define 4 constructors out of which the be below one will be used for simplicity:

Exception(String msg)

This form lets the user to specify the description of the Exception.

Sample program:

```
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
      // calling the constructor of parent Exception
      super(str);
   }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

   // method to check the age
   static void validate (int age) throws InvalidAgeException{
     if(age < 18){
```

```
        // throw an object of user defined exception
        throw new InvalidAgeException("age is not valid to vote");
    }
      else {
       System.out.println("welcome to vote");
       }
     }

    // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException object
            System.out.println("Exception occured: " + ex);
        }

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

## Output:

```
Caught the exception
Exception occured: exception_samples.InvalidAgeException: age is not valid
to vote
rest of the code...
```