Exception Handling

10.1 Definiton

An exception is an event that occurs during the execution of a program that disturbs the normal flow of instructions

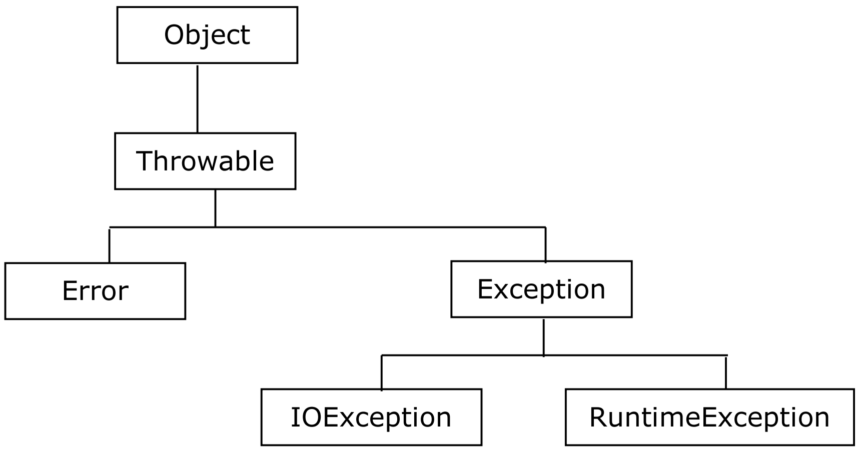
Exceptions are erroneous events like **Divide by Zero**, **Number Format Exception, Negative Array Size, Opening a File that Does Not Exist** etc.

A Java exception is an object describes an exceptional condition that has occurred in a piece of code. Error handling becomes necessary when you develop applications that need to cause of unexpected situations.

Advantages

1. **Separating Error Handling code from "Regular" code**
2. **Grouping the possible Error types**

Class Hierarchy



10.2 Types of Exceptions

10.2.1 Arithmetic Exception

This exception is generated when the divisor is zero

```
class DivisionDemo
{
	public static void main(String args[])
	{
		int a = Integer.parseInt(args[0]);
		int b = Integer.parseInt(args[1]);
		int c = a/b;
		System.out.println("Division of two numbers is "+c);
		System.out.println("End of program");
	}
}
```

When a java run-time system (JVM) detects the attempt to divide by zero, it constructs a new exception object and then `throws` this exception. This cause the execution of above program to stop, because once an exception has been throwns, it must be caught by an exception handler and dealt with immediately.

In the example, we haven't supplied any exception handlers of our own, so the exception is caught by the **default handler** by the java-runtime system.

Any exception that is caught by your program will ultimately processed by the default handler. Hence default handler display a string describing the exception,

```
java.lang.ArithmeticExcepiton : / by zero
```

Using try and catch

Although the default exception handler provided by the java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits.

- **First, it allows you fix the error.**
- **Second, it prevent the program from automatically terminating**

To guard against a Run-Time Exception, simply enclose the code that you want to monitor inside a `try` block. Immediately following the `try` block, include a `catch` clause that specifies the exception type that you wish to catch.

Program //demonstration of arithmetic exception

```
class DivideException
{
      public static void main(String args[])
      {
            int a,b,c;
            try
            {
                  a=Integer.parseInt(args[0]);
                  b=Integer.parseInt(args[1]);
                  c=a/b;
                  System.out.println("Division of two nos is "+ c );
            }
            catch(ArithmeticException e)
            {
                  System.out.println("Cannot divide by zero");
            }
            System.out.println("End of program");
      }
}
```

OUTPUT

```
C:\>  Java DivideException 100 0
      Cannot divide by zero
```

Program //another example for divide by zero exception

```
import java.util.Random;
class HandleError
{
      public static void main(String args[])
      {
            int a=0, b=0, c=0;
            Random r = new Random();

            for(int i=0; i<10; i++)
            {
                  try
                  {
                        b = r.nextInt();
                        c = r.nextInt();
                        a = 12345 / (b/c);
                  }
                  catch (ArithmeticException e)
                  {
                        System.out.println("Division by zero.");
                        a = 0; // set a to zero and continue
                  }
            System.out.println("a: " + a);
            }
       }
}
```

10.2.2 ArrayIndexOutOf BoundsException

This exception is generated when an array index is invalid.

Program //demonstration of ArrayIndexOutOfBoundsException

```
class ArrayException
{
      public static void main(String arg[])
      {
            try
            {
                  int a[]=new int[5];
                  a[0]=25;
                  a[1]=30;
                  a[2]=14;
                  a[3]=15;
                  a[4]=100;
                  System.out.println(a[0]);
                  System.out.println(a[1]);
                  System.out.println(a[2]);
                  System.out.println(a[3]);
                  System.out.println(a[4]);
                  System.out.println(a[5]);
            }
            catch(ArrayIndexOutOfBoundException e)
            {
            System.out.println("Cannot access the value of a[5]" + e);
            }
      }
}
```

OUTPUT

```
c:\> java ArrayException
25
30
14
15
100
Cannot access the value of a[5]
```

10.2.3 NegativeArraySize Exception

This exception is generated when an array size is negative.

Program //demonstration of NegativeArraySizeException

```
class NegativeException
{
      public static void main(String args[])
      {
            try
            {
                  int a[]=new int [-5]

            }
            catch(NegativeArraySizeException e)
            {
            System.out.println("Array size should be positive integer");
            }
      }
}
```

OUTPUT

```
c:\> java NegativeException
Array size should be a positiveinteger
```

10.2.4 NumberFormat Exception

This exception is generated when the string parsed is not a number.

Program //demonstration of NumberFormatException

```
class NumberException
{
      public static void main(String args[])
      {
            try
            {
                  int a,b,c;
                  a=Integer.parseInt(args[0]);
                  b=Integer.parseInt(args[1]);
                  c=a/b;
                  System.out.println(c);
            }
            catch(NumberFormatException e)
            {
            System.out.println("Argument should be numeric values");
            }
      }
}
```

OUTPUT

```
c:\>java NumberException 100 o
Argument should be numeric values
```

Other Exceptions

10.2.5 FileNotFoundException

It is generated when a requested file is not found
.

10.2.6 SQLException

It is generated when error occurs during a database access.

10.2.7 UnKnownHostException

It is generated when the host (web site) cannot be located.

10.4 Multiple Catch block

More than one exception should be raised by a single piece of code. To handle this type of situation, you can specify two or more catch blocks, each catch as different types of exception. When an exception is thrown, each catch statement is expected in order to print the messages. If the exception does not find a matching catch block, it is passed to the default handler.

```
class MultiCatch
{
	public static void main(String args[])
	{
		try
		{
			int a=Integer.parseInt(args[0]);
			int b=Integer.parseInt(args[1]);
			int c=a/b;
			System.out.println("Result = "+c);
		}
		catch(ArithmeticException e)
		{
			System.out.println("Zero divide error");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("Enter minimum two values");
		}
		catch(NumberFormatException e)
		{
			System.out.println("Enter only numeric values ");
		}
}
```

OUTPUT

```
c:\>java MultiCatch 100
Enter minimum two values
```

Note Instead of having multiple catch block, we can use single catch block by specifying simply Exception.

```
class SimpleException
{
	try
	{
		exception cause coding...;
	}
	catch (Exception e)
	{
		System.out.println("Exception is "+e);
	}
}
```

10.5 Throw

It is possible to throw an exception explicitly using throw statement.

Syntax

```
throw new Exception-name(String);
```

```
class ThrowDemo
{
      public static void main(String args[])
      {
            try
            {
                  int a,b,c;
                  a=Integer.parseInt(args[0]);
                  b=Integer.parseInt(args[1]);

                  if(b==0)
                        throw new ArithmeticException();

                        c=a/b;
                        System.out.println("Division of two nos is "+ c);
            }
            catch(ArithmeticException e)
            {
                  System.out.println("Cannot divide by zero ");
                  return;
            }
      }
}
```

OUTPUT

```
c:\>java ThrowDemo 100 0
Cannot divide by zero
```

10.6 Throws

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

Syntax

```
      return-type method_name(parameter-list) throws exception-list
      {
            //body of method
      }
```

Program

```
class Throws
{
      public static int div(int a,int b)throws ArithmeticException
      {
            int c;
            c=a/b;
            return c;
      }
}
class ThrowsDemo
{
      public static void main(String args[])
      {
            Throws t = new Throws();
            try
            {
                  int d=t.div(20,0);
            }
          catch(ArithmeticException e)
          {
            System.out.println("Cannot Divide by zero");
          }
      }
}
```

OUTPUT

```
c:\>java ThrowsDemo 100 0
Cannot divide by zero
```

10.7 Finally

The **finally** block ***always*** executes when the **try** block exits. This ensures that the **finally** block is executed even if an unexpected exception occurs. But **finally** is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a **return, continue, or break**. Putting cleanup code in a **finally** block is always a good practice, even when no exceptions are anticipated.

When an exception is generated in program, control will be goto catch block and it will stop the process sometimes it may be necessary to the program to stop certain activities before termination.

Note : The finally block will be after try, catch or after try.
finally block is optional

```
class FinallyDemo
{
      public static void main(String args[])
      {
            int a=0,b=0,c=0;
            try
            {
                  a=Integer.parseInt(args[0]);
                  b=Integer.parseInt(args[1]);
                  c=a/b;
                  System.out.println("Division of two nos is "+c);
            }
            catch(ArithmeticException e)
            {
                  System.out.println("Cannot divide zero ");
            }
            finally
            {
                  System.out.println("finally block will execute before
                  termination..");
            }
                  System.out.println("End of program ");
      }
}
```

OUTPUT

```
c:\>java FinallyDemo 100
```

```
finally block will execute before termination..
Exception in Thread "main" ArrayIndexOutOfBoundsException: 0
```

10.8 User-Define Exception

Customized exceptions are necessary to handle abnormal conditions of applications created by the user. The advantage of creating such an exception class is that, according to situation defined by the user an exception can be thrown.

Steps to create User-Defined Exception

1. Create a subclass of Exception.
2. Inside the subclass define the constructor.
3. From the program throw the above exception.

Program

```
import java.lang.*;

class DivisionException extends Exception
{
      public DivisionException ()
      {
            System.out.println("DivisorError, Cannot divide by zero ");
      }
      public DivisionException (String s)
      {
            System.out.println(s);
      }
}
class MyExceptionDemo
{
      public static void main(String args[])
      {
            try
            {
                  int a,b,c=0;
                  a=Integer.parseInt(args[0]);
                  b=Integer.parseInt(args[1]);

                  if(b==0)throw new DivisionException();

                  else
                  {
                        c=a/b;
                        System.out.println("Division of two nos is "+ c);
                  }
            }
            catch(DivisionException e)
            {
                  System.out.println(e);
            }
      }
}
```

OUTPUT

```
c:\>java MyExceptionDemo 100 0
DivisorError, Cannot divide by zero
```