Chapter - 8

Exceptional Handling

Exceptional Handling

Introduction

Generally errors are raised at two different situations i.e. compile time or run time.

The errors that are raised at compilation time are known as compile time errors.

Example: Syntax errors

The errors that are raised at runtime are known as runtime errors.

Example: because of invalid input.

Runtime error is known as **Exception**. *An exception is an abnormal condition that is raised in a code sequence at runtime.*

What happens when an exception occurs?

When an exception(runtime error) occurs while executing the program, the program stops execution unconditionally and displays the error messages. This type of termination of program is known as **abnormal program termination**.

Dis-advantages of abnormal program termination

Loss of data: When an exception raises, the data given as input to the program and data in the opened files in the program are lost.

A program to demonstrate abnormal program termination

Bin>edit exp1.java

```
class exp1
{
      public static void main(String argv[])
      {
            int a,b,c;
            a=Integer.parseInt(argv[0]);
            b=Integer.parseInt(argv[1]);
            c=a/b;
            System.out.print("\n c="+c);
            System.out.print("\n program terminates.....");
      }
}
```

```
Bin>javac exp1.java
Bin>java exp1 10 2
        c=5
        program terminates.....

Bin>java exp1 12 0
        Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exp1.main(exp1.java:8)
```

Explanation:

In the first execution

Bin>java exp1 10 2

passes **10** and **2** to the **argv[]** of main() method and these values are stored in the form of strings. In the main() these are parsed to integers and are stored in **a** and **b** variables therefore **a** is stored with **10** and **b** with **2**.

The statement **c=a/b;** calculates and the result **5** is assigned with **c** and it printed on the monitor as shown above. There is no error for this execution.

In the second execution

Bin>java exp1 12 0

passes **12** and **0** to the **argv[]** of main() method and these values stored in the form of strings. In the main() these are parsed to integers and are stored in **a** and **b** variables therefore **a** is stored with **12** and **b** with **0**.

The statement **c=a/b;** calculates **12/0** which is infinite and jvm identifies it as an error therefore the jvm creates an object of exception type (in this case it is ArithmeticException) and error messages are stored into the object and also pushes the same into system stack. Finally the program is terminated and error messages are popped from stack and prints on monitor. This type of termination is known as abnormal program termination. The disadvantage of abnormal program termination is **loss of data**.

To overcome the above disadvantage exceptional handling mechanism can be used.

Exceptional handling is a mechanism, using which runtime errors can be handled so that the program is terminated normally and no loss of data.

Exceptional handling can be achieved in Java using the keywords **try**, **catch**, **throw**, **throws** and **finally**.

Syntax:

```
try
{
        : // code to scan for errors
        :
}
catch(exceptiontype1 ex1)
{
        : // code to handle exceptiontype1
}
catch(exceptiontype2 ex2)
{
        : // code to handle exceptiontype2
        :
}
:
:
finally
{
        :// final code to execute
        :
}
```

A try can have any number of catch blocks and catch blocks should immediately follow the try block. Each catch block can handle one type of exception that is raised in try block.

try block

The **try** block is used to scan the code for exception during the execution of the program. If an exception occurs, the program flow is interrupted and the control goes to **catch** blocks. The **try** block is used to test the code for an exception.

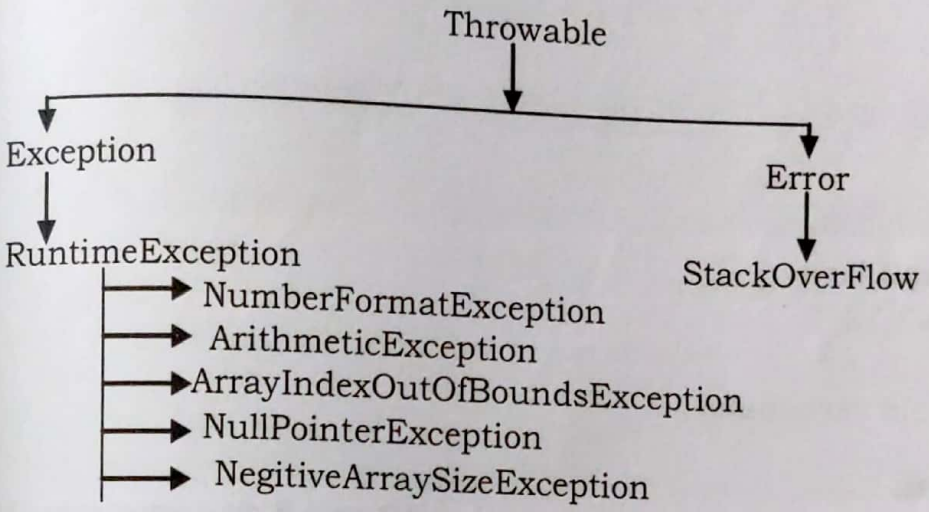
catch block

The exception raised in the **try** block is caught by the **catch** block and handles it so that the program does not terminate abnormally. The **catch** block should follow immediately after try block and can have multiple **catch** blocks

finally block

The **finally** block executes if an exception is raised or not raised in the **try** block. It executes before the method is terminated from the **try** block because of any abnormal situations. The **finally** block should be the last block of **try**. A **try** can contain any number of **catch** blocks but should contain only one **finally** block. The try should contain either **catch** or **finally** or both.

Exceptions are predefined classes in java which are present in **java.lang** package. The super most class of exception classes is **Throwable.** This class is further inherited into **Exception** and **Error** classes. The **Exception** class is further inherited into **RuntimeException**. The **RuntimeException** is having subclasses and they can be seen in the following hierarchy.



The **Exception** and its subclasses exceptions are raised because of problems in the java program. These exceptions can be handled in the java program using the exceptional handling mechanism so that program does not terminate abnormally, but terminate normally.

The **Error** type of exceptions are raised because of problems in the jvm which is outside the program. These Exceptions are not related to java program therefore the jvm itself take care of **Error** type of exception and we need not to write any exceptional handling code. The **StackOverflow** is an example of **Error** type of exception.

A program demonstrating ArithmeticException

Bin>edit exp2.java

```
class exp2
{
```

```
    public static void main(String argv[])
    {
          int a,b,c;
          a=Integer.parseInt(argv[0]);
          b=Integer.parseInt(argv[1]);
          try
          {
                c=a/b;
                System.out.print ("\n c="+c);
          }
          catch(ArithmeticException e)
          {
             System.out.print("\n Error:Divide by zero");
             System.out.print("\n e : "+e);
          }
          System.out.print("\n program terminates…");
    }
}
```

Bin>javac exp2.java
Bin>java exp2 12 3

c=4
program terminates...

Explanation:

To the **exp2** program we are passing **12** and **3** as arguments, **12** stores into **argv[0]** and **3** stores into **argv[1]**

Bin>javac exp2 15 0

Error: Divide by zero
e = java.lang.ArithmeticException: / by zero
program terminates...

Explanation:

To the main() method **15** and **0** are passed to **argv[]** where **argv[0]** is stored with **15** and **argv[1]** is stored with **0**. In the main(), **argv[0]** is assigned to **a** i.e **15** and **argv[1]** is assigned to **b** i.e **0**. At the statement **c=a/b;** where **15** is divisible by **0** which is an error therefore the jvm creates an object of **ArithmeticException** class and the object is stored with error messages. Finally the jvm throws an arithmetic exception object which is caught by **catch** block and the exception object address is assigned to **e** and the catch block is executed where we get the output as shown above. After execution

of **catch** block the remaining statements after the **catch** block are executed till it reaches main and the program terminates normally.

A program to demonstrate ArrayIndexOutOfBoundsException

Bin>edit exp3.java

```
class exp3
{
   public static void main(String argv[])
   {
      try
      {
         System.out.print("\n first argument="+argv[0]);
         System.out.print("\n second argument="+argv[1]);
         System.out.print("\n third argument="+argv[2]);
      }
      catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.print("\n Index out of range.");
         System.out.print("\n Error : "+e);
      }
      System.out.print("\n program terminates…");
   }
}
```

Bin>javac exp3.java

Bin>java exp3 10 20

first argument=10
second argument=20
Index out of range.
Error : java.lang.ArrayIndexOutOfBoundsException: 2
program terminates...

Explanation:

When the above program is executed with **java exp3 10 20** the arguments **10** and **20** are passed to **argv[]** of main, therefore **argv[0]** is stored with **10** and **argv[1]** is stored with **20**. In the main(), the statements

System.out.print("\n first argument="+argv[0]); argv[0] prints **10** and we get the output **first argument=10**

System.out.print("\n second argument="+argv[1]); **arg[1]** prints **20** and we get the output **second argument=20**

In the statement **System.out.print("\n third argument="+argv[2]);** the array argv does not contain index **2** therefore the jvm identifies as error and creates the object of **ArrayIndexOutOfBoundsException** and throws it, which is caught by catch block and prints the messages

Index out of range.
Error : java.lang.ArrayIndexOutOfBoundsException: 2

After executing the **catch** block the next statement prints the message **program terminates...** and finally the program terminates at the end of main().

Bin> java exp3 10 20 30

first argument=10
second argument=20
third argument=30
program terminates...

Explanation

When the above program is executed with **java exp3 10 20 30**, the arguments **10**, **20** and **30** are passed to **argv[]** of main, therefore **argv[0]** is stored with **10**, **argv[1]** is stored with **20** and **argv[2]** is stored with **30**. In the **try**, the three statements prints

first argument=10
second argument=20
third argument=30

As no exception is raised in the **try** block, the control comes out of catch block where it prints the message **program terminates...** and finally the main() terminates.

A program to demonstrate NegitiveArraySizeException.

Bin>edit exp4.java

```
class exp4
{
   public static void main(String argv[ ])
   {
```

```
        int a[],n,i;
        try
        {
              n=Integer.parseInt(argv[0]);
              a=new int[n];
              for(i=0; i<n; i++)
              {
                    a[i]=(i+1)*10;
                    System.out.print("\t"+a[i]);
              }
        }
        catch(NegitiveArraySizeException e)
        {
              System.out.print("\nError : "+e);
        }
        System.out.print("\n program terminates...");
  }

}
```

Bin>javac exp4.java
Bin>java exp4 3
10 20
program terminates...
Bin>java exp4 -2
Error : java.lang.NegitiveArraySizeException :
program terminates...

Explanation:

When the above program is executed using **java exp4 -2** , **-2** passes to **argv[0]** element of main(). In the **try** block, at the statement **n=Integer.parseInt(argv[0]);** the **parseInt()** method converts **argv[0]** value "**-2**" string into integer and assigns to **n**.

At the statement, **a=new int[n];** raises an **Negitive Array Size Exception** because **n** is negitive value and negitive size memory can not be allocated as an array. This exception is caught by the **catch** block and prints the error message as shown in above output. The remaining statements after **catch** block executes and finally program terminates.

A program to demonstrate NullPointerException

Bin> edit exp5.java

```
class sample
{
   public int x=10;
}
class exp5
{
   public static void main(String argv[])
   {
         sample obj[ ];
         try
         {
               obj=new sample[2];
               System.out.print("\n obj[0].x="+obj[0].x);
               System.out.print("\n obj[1].x="+obj[1].x);
         }
         catch(NullPointerException e)
         {
               System.out.print("\n Error : " + e );
         }
         System.out.print("\n program terminates...");
   }
}
```

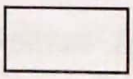
Bin>javac exp5.java

Bin>java exp5

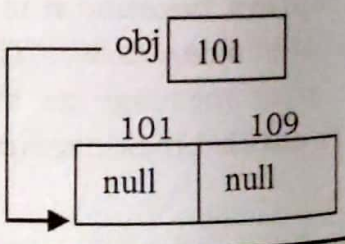
java.lang.NullPointerException:
program terminates...

Explanation:

When the above program is executed, in the main() method, the statement creates a sample **obj[]**; reference element of an array of reference elements of **sample** type.i.e.,



the statement, **obj=new sample[2];** allocates 2 reference elements of sample type i.e.,



The reference elements are initialized with **null**.

The statement **System.out.print("\n obj[0].x="+obj[0].x);** generates error, because **obj[0]** element is stored with **null** and using **null** we can't access any member. i.e. **obj[0].x** becomes **null.x** which generates a **NullPointerException** and the jvm throws **NullPointerException** which is caught by the catch block and executes the statement **System.out.print("\n Error : " + e);** which prints the output as shown in the above output.

To overcome the above exception we have to allocate memory of sample class to the reference elements.

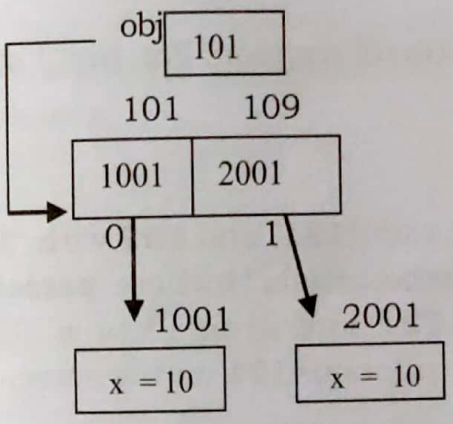
Example:

```
try
{
      obj=new sample [2];
      obj[0]=new sample();
      obj[1]=new sample();
      System.out.print("\n obj[0].x="+obj[0].x);
      System.out.print("\n obj[1].x="+obj[1].x);

}
catch(NullPointerException e)
{
      System.out.println("\n Error : "+ e );

}
```

In the above code each reference element of array is allocated with the memory of example class i.e.



Therefore the statements:

System.out.print("\n obj[0].x="+obj[0].x);
System.out.print("\n obj[1].x="+obj[1].x);

prints **obj[0].x** and **obj[1].x** as **10**

A program to demonstrate NumberFormatException

Bin>edit exp6.java

```
class exp6
{
   public static void main(String argv[])
   {
         String str1="121", str2="15A";
         int n, m;
         try
         {
               n=Integer.parseInt(str1);
               System.out.print("\n n="+n);
               m=Integer.parseInt(str2);
               System.out.print("\n m="+m);

         }
         catch(NumberFormatException e)
         {
               System.out.println("\n Error : "+ e);

         }
         System.out.println("\n program terminates...");
   }
}
```

Bin>javac exp6.java

Bin>java exp6

```
n=121
Error :java.lang.NumberFormatException: For input string : "15A"
 program terminates....
```

Explanation :

In the main(),**str1** is stored with "**121**" and **str2** with "**15A**".In the try block the statement **n=Integer.parseInt(str1);**where **parseInt()** converts "**121**" (str1) into integer value **121** and assigns to **n**. The statement **System.out.println("\n n="+n);** prints **n=121** on the screen.

In the statement **m= Integer.parseInt(str2);** the **parseInt()** can't convert "**15A**"(str2) into integer format, as a result the JVM generates a **NumberFormatException** and throws it . The thrown Exception is caught by the **catch** block where it prints the message.

Error: java.lang.NumberFormatException:

After the execution of **catch** block, the print statement executes and finally main() terminates.

Multiple Catch Blocks

A try can have any number of catch blocks and each catch block can handle only one exception.

Syntax:

```
try
{
        //code to scan for exceptions
}
catch(Exceptiontype1 eobj1)
{
        //code to handle Exceptiontype1
}
catch(Exceptiontype2 eobj2)
{
        //code to handle Exceptiontype2
}
.
.
catch(ExceptiontypeN eobjN)
{
        //code to handle ExceptiontypeN
}
finally
{
        //find code to execute
}
```

A program to demonstrate multiple catch blocks.

Bin>edit mcatch.java

```
class mcatch
{
   public static void main(String argv[])
   {
        int a=0,b=0,c=0;
        try
        {
             a=Integer.parseInt(argv[0]);
             b=Integer.parseInt(argv[1);
             c=a/b;
             System.out.println("\n c="+c);
        }
```

```
        catch(NumberFormatException e1)
        {
              System.out.print("\n Error1 : "+e1);
        }
        catch(ArithmeticException e2)
        {
             System.out.print("\n Error2 : "+e2);
        }
        catch(IndexOutOfBoundsException e3)
        {
             System.out.println("\n Error3 : "+e3);
        }
        System.out.println("\n program terminates...");
    }
}
```

Bin>javac mcatch.java
Bin>java mcatch 10 2A

Error1 : java.lang.NumberFormatException: For input string : "2A"
program terminates...

Explanation:

When the above program is executed with the statement **java mcatch 10 2A** calls main() and passes "**10**" and "**2A**" to the **argv[]**. Where **argv[0]** is stored with "**10**" and **argv[1]** is stored with "**2A**". The statement, **b=Integer.parseInt(argv[1]);** raises a **NumberFormatException** because **argv[1]** is stored with "**2A**" which is not a valid integer value. This Exception is caught by the first catch block and which handles it and terminates all catch blocks and remaining statements executes and finally main terminates.

Bin>java mcatch 10 0

Error2 : java.lang.ArithmeticException: / by zero
program terminates ...

Explanation:

When the above program is executed with the statement **java mcatch 10 0** calls main() and to which passes "**10**" and "**0**" to the **argv[]** of main(). Where **argv[0]** is stored with "**10**" and **argv[1]** is stored with "**0**". In the main() the parse statement assigns **10** to **a** and **0** to **b**. The statement **c=a/b;** raises an **ArithmeticException** because **a**(10) divisible by **b**(0) is infinity.

This exception is caught by the second **catch** block which handles the exception and terminates all catch blocks. The remaining statements execute and finally main terminates.

Bin> java mcatch 12

```
Error3 : java.lang.ArrayIndexOutofBoundException:1
program terminates...
```

Explanation:

When the above program is executed with the statement **java mcatch 12** passes "**12**" to **argv[0]**. In the main() the statement **b=Integer.parseInt(argv[1]);** raises **ArrayIndexOutOfBoundsException** because index number **1** is not the limitations of array **argv[]**. This exception is caught by the third **catch** block which handles the exception and we get output as shown above.

Handling Multiple Exceptions

It is possible to handle all exceptions by a single catch block. A catch block with **Exception** type can handle all runtime exceptions because runtime exceptions are the subclasses of **Exception** class. A super class reference variable can store the reference of its subclasses.

Syntax:

```
try
{
   :
}
catch(Exception e)
{
   //handles all runtime exception
}
```

Bin>edit allexp.java

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

```
                System.out.print("\nc="+c);
            }
            catch(Exception e)
            {
                System.out.print("\nError : "+e);
            }
            System.out.print("\nprogram terminates...");
        }
    }
```

Bin>javac allexp.java

Bin>java allexp 12 2A

Error : java.lang.NumberFormatException:"2A"
program terminates...

Bin>java allexp 15 0

Error : java.lang.ArithmeticException: / by zero
program terminates...

Bin>java allexp 12

Error : java.lang.ArrayIndexOutOfBoundsException: 1
program terminates...

Bin>java allexp 15 3

c=5
program terminates...

When a catch block with **Exception** type is used with other catch blocks then the catch block with **Exception** type should be the last catch block of try.

For Examp le:

```
try
{
        :
}
catch(NumberFormatException e1)
{
        :
}
catch(ArthemeticException e2)
{
        :
}
```

```
catch(Exception e3)
{
        :
}
```

In the above example, the catch with **Exception** type handles all exceptions other than **NumberFormatException** and **ArithmeticException**. But the following is not correct.

```
try
{
        :
}
catch(Exception e1)
{
        :
}
catch(ArthemeticException e2)
{
        :
}
```

In the above program all exceptions are handled by first catch block **catch(Exception e1)** and does not reach to **catch(ArthemeticException e2)**. Therefore we should not use catch blocks after the catch block with **Exception** type.

Nested try

A **try** can be written in another **try** and it is known as **nested try**. Exceptions raised in outer block can be handled by outer try catch blocks where as exceptions raised in the inner try can be handled by inner try catch blocks otherwise the exceptions are reached to outer try catch block and is handled.

Syntax:

```
try
{
    :
    try
    {
        :
    }
    catch(Exceptiontype e1)
    {
        :
    }
    :
}
catch(Exceptiontype e2)
{
        :
}
:
```

A program demonstrating nested try

Bin>edit nesttry.java

```
class nesttry
{
   public static void main(String argv[] )
   {
          int a=0,b=0,c=0;
         try
         {
               a=Integer.parseInt(argv[0] );
               try
               {
                     b=Integer.parseInt(argv[1] );
                     c=a/b;
                     System.out.print("\nc= "+c);
               }
               catch(ArithmeticException e1)
               {
                     System.out.println("\nError1 : "+e1);
               }
         }
         catch(NumberFormatException e2)
         {
               System.out.println("\nError2 : "+e2);
         }
         catch(ArrayIndexOutOfBoundsException e3)
         {
               System.out.println("\nError3 : "+e3);
         }
         System.out.println("\n program terminates...");
   }
}
```

Bin> javac nesttry.java

Bin>java nesttry 10A 2

Error2 : java.lang.NumberFormatException: For input string: "10A"
program terminates...

Explanation:

In the above execution "**10A**" stores into **argv[0]** and "**2**" stores into **argv[1]**. In the outer try the statement **a=Integer.parseInt(argv[0])**; raises

a **NumberFormatException** because the value of **argv[0]** "**10A**" is not a valid integer value. This Exception is caught by the outer try catch block and handles it.

Bin>java nesttry 12 3A

Error2 : java.lang.NumberFormatException: For input string: "3A"
program terminates...

Explanation:

In the above execution "**12**" and "**3A**" stores into **argv[0]** and **argv[1]**. In the inner try the statement **b=Integer.parseInt(argv[1]);** raises a **NumberFormatException** because the value of **argv[1]** (3A) is not a valid integer. This exception cannot be handled by the inner try block therefore the exception is handled by outer try catch block.

Bin>java nesttry 15 0

Error1 : java.lang.ArithmeticException : / by zero
program terminates...

Explanation:

In the above execution **argv[0]** is stored with **15** and **argv[1]** with **0** which are converted into integers and stored into **a**, **b**. The statement **c=a/b;** raises an **ArithmeticException** because **a**(15) divided with **b**(0).This exception is caught by the inner try catch block and handles it.

Bin>java nesttry 20

Error3 : java.lang.ArrayIndexOutOfBoundsException: 1
program terminates...

Explanation:

In the above execution **20** is passed into **argv[0]**. In the inner try block **b=Integer.parseInt(argv[1]);** raises an **ArrayIndexOutOfBoundsException** because **argv[1]** index does not exist. This exception can not be handled by inner try catch block, therefore the outer try catch block handles it.

Bin>java nesttry 12 3

c=4
program terminates...

Throwing Exception Manually

In the previous program exceptions are thrown by the JVM internally based on the type of exception raised. Java also allows to throw the exceptions

explicitly in the program. Exceptions can be thrown explicitly by using the throw keyword.

Syntax:

```
throw ThrowableInstance;
```

In the Syntax **ThrowableInstance** means it should be an object of **Throwable** or its subclasses. The **throw** keyword throws the exception object explicitly.

Example:

Bin>edit throw1.java

```
class throw1
{
  public static void main(String argv[ ])
  {
    int a=0,b=0,c=0;
    try
    {
        a=Integer.parseInt(argv[0]);
        b= Integer.parseInt(argv[1]);
        if(b==0)
          throw new ArithmeticException("divide by zero");
        c=a/b;
        System.out.print( "\n c= "+c);
    }
    catch(ArithmeticException e)
    {
        System.out.println("\nError : "+ e);
    }
    System.out.println("\n program terminates...");
  }
}
```

Bin>javac throw1.java
Bin>java throw1 12 4

```
    c=3
  program terminates...
```

Bin>java throw1 12 0

```
    Error : java.lang.ArithmeticException: divide by zero
    program terminates...
```

Explanation:

In the above execution **12** and **0** are passed to **argv[0]** and **argv[1]**. In the try **12** and **0** are assigned to **a** and **b** respectively. In the if condition **b==0** gets **true** and the statement **throw new ArithmeticException("divide by zero");** is executed. In this statement the new keyword creates an object of **ArithmeticException** class and the object is stored with error message "**divide by zero**" and the object is thrown by the throw keyword. This thrown object is caught by the catch block and prints the error message **Error: divide by zero error.** After execution of catch block finally the **program terminates...** message prints on monitor and the program terminates.

Re-throw

Re-throw is the concept of throwing an exception more than once. For example consider the following example

Bin>edit rethrow1.java

```
class myclass
{
   public void divi(int x, int y)
   {
         int z=0;
         try
         {
               if(y==0)
         throw new ArithmeticException("divide by zero");
               z=x/y;
               System.out.println("\n z="+z);
         }
         catch(ArithmeticException e)
         {
               System.out.print("\n Error :"+e);
               throw e;
         }
   }
}
class rethrow1
{
   public static void main(String argv[ ])
   {
         myclass obj=new myclass();
         try
         {
```

```
                obj.divi(10,2);
                obj.divi(12,0);
            }
            catch(ArithmeticException e)
            {
                System.out.print("\n recaught:"+e);
            }
            System.out.print("\n program terminates...");
    }
}
```

Bin>javac rethrow1.java
Bin> java rethrow1

```
z=5
java.lang.ArithmeticException: divide by zero
recaught:java.lang.ArithmeticException: / by zero
 program terminates...
```

Explanation:

In the statement obj.**divi(12,0);**calls the method **divi(int x,int y)**. In the try block the condition **y==0** gets satisfies and throws the **ArithmeticException**, which is caught by catch block. The catch after handling the exception it again throws the same exception using throw keyword and it is known as **rethrow**. This throw exception is send to the main method from where the **divi()** methods is called it is handled by catch block of main.

The finally block

A **try** should contain either **catch** or **finally** block. The try can contain any number of catch blocks but only one finally block. When an exception raises in try block the concerned catch block executes otherwise the catch block does not executes, where as finally block executes if an exception raises or not or in any other abnormal situations i.e. the statements in finally block executes compulsorily.

Syntax:

```
try
{
        :
        : // code to scan for exceptions
        :
}
```

```
catch(Exceptiontype1 e1)
{
        : //code to handle Exceptiontype1
}
catch(Exceptiontype2 e2)
{
        : //code to handle Exceptiontype2
}
:
finally
{
    : // final code to execute here
}
```

Example:

Bin>edit finally1.java

```
class sample
{
  public void A()
  {
    System.out.print("\n A() executing..");
    try
    {
      throw new ArithmeticException(" divide by zero");
    }
    catch(ArithmeticException e)
    {
      System.out.print("\n ErrorA :" + e);
    }
    finally
    {
      System.out.print("\n Finally of A..");
    }
    System.out.print("\n A() terminates..");
  }

  public void B()
  {
    System.out.print("\n B() executing..");
    try
    {
```

```
        System.out.print("\n No exception in B()..");
        }
        catch(ArithmeticException e)
        {
          System.out.print("\n ErrorB :" + e);
        }
        finally
        {
          System.out.print("\n Finally of B..");
        }

        System.out.print("\n B() terminates..");
    }
    public void C(int x)
    {
        System.out.print("\n C() executing..");
        try
        {
          if(x==1)
            return;
        }
        finally
        {
          System.out.print("\n Finally of C..");
        }
        System.out.print("\n C() terminates.."); //this
                        // statement does not execute
    }
}
class finally1
{
    public static void main(String argv[ ])
    {
        sample obj=new sample();
        obj.A();
        obj.B();
        obj.C(1);
    }
}
```

```
Bin>javac finally1.java
Bin> java finally
A() executing..
ErrorA : java.lang.ArithmeticException: divide by zero
Finally of A..
A() terminates..
B() executing..
No execption in B()..
Finally of B..
B() terminates..
C() executing..
Finally of C..
```

Explanation:

In the above program, at the statement **obj.A();** the method **A()** is called where the print statement prints **A() executing..** and in the try it throws ArithmeticException which is caught by the catch block and prints the error message **ErrorA: divide by zero** and then the finally block executes where it prints **Finally of A...** After the finally block the print statement prints **A() terminates**. In this case, as an exception is raised the catch block executed and also finally.

At the statement **obj.B();** the method **B()** is called where the print statement prints **B() executing..** and in the try block it prints the message **No exception in B()...** As no exception is raised in B() the catch block does not executes where the finally block executes where it prints **Finally of B...** After the finally block the print statement prints **B() terminates...**

At the statement **obj.C(1);** the method **C(int x)** is called where **x** is passed with **1**, the print statement prints **C() executing..** and in the try block the condition **x==1** gets true and **return** statement executes which terminates the method. But before termination of **C()** the finally block is executed where it prints **C() terminates...**

In this way the finally block is executed if an exception is raised or not or in any abnormal situations. Therefore any code that to be executed compulsorily should be written in finally block, for example if a file is opened on entry of method and want to close the file in any situation that raises in the method, then the code of closing the file can be written in finally block and so on.

The **finally** block of **try** is optional.

Types of Exceptions

All exception classes are defined in **java.lang** package which is the default package in java. Most of these exceptions are the subclasses of **RuntimeException** class. These exception classes are of two types such as checked exceptions and unchecked exceptions.

Unchecked Exceptions

All java methods by default have permissions to throw unchecked exceptions. When the java complier come across an unchecked exception in a method then the compiler does not check any condition with it whether the method can throw the exception or not and therefore called unchecked exception. The unchecked exceptions need not to be included in any method's throws list. The unchecked exception classes are listed in the below table. All the above programs are the example programs on unchecked exceptions.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a mehod.
IllegalMonitorsStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Java's Unchecked RuntimeException Subclasses

Checked Exceptions

The java methods do not have permissions to throw the checked exceptions by default. When the compiler comes across the checked exception in the method, it checks the condition whether the method can throw the exception or not therefore called checked exception. The checked exceptions need to be included in the method's throws list otherwise complier gives an error. The checked exception classes are listed in the below table.

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Java's Checked Exceptions Defined in java.lang

A program to demonstrate checked exception

Bin>edit checkexp.java

```
class checkexp
{
  public static void main(String argv[]) throws
                        ClassNotFoundException
   {
        System.out.print("\nSample program on
                        checked exception.");
        try
        {
              throw new ClassNotFoundException("class
                                        not found");
        }
        catch(ClassNotFoundException e)
        {
```

```
            System.out.print("\n Error :"+e);
        }
        System.out.print("\nProgram terminates.");
        }
}
```

Bin>javac checkexp.java
Bin>java checkexp

```
Sample program on checked exception.
Error : java.lang.ClassNotFoundException : class not found
Program terminates.
```

Explanation:

In the above program, the **try** block is throwing a checked exception **ClassNotFoundException**, therefore this exception is listed using the throws keyword at the main method header. If the main method is not declared with throws keyword then the compile generates error. When the above program is executed we get the output as shown above.

The toString() method

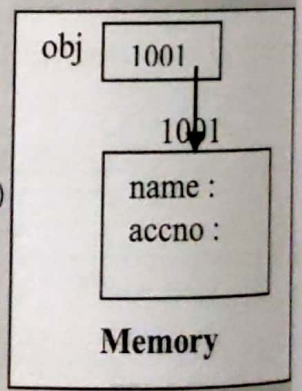
The **toString()** method whose signature is predefined and it can be overridden in any class. This method is automatically executed when an object is used in the expression. This method can also be called using the object name.

Syntax:

```
public String toString()
{
        :
}
```

A program to demonstrate the toString() method.
Bin>edit tostring1.java

```
class bank
{
    private String name;
    private int accno;
    public void setbank(String n,int a)
    {
        name=n;
        accno=a;
    }
```



```
    public String toString()
    {
      .    return  "name="+name+"\t accno="+accno;
    }
}
class tostring1
{
      public static void main(String argv[])
      {
            bank obj=new bank();
            obj.setbank("kumar",105);
            System.out.print("\nBank :" +obj);
            String str="Bank Details :" +obj;
            System.out.print("\n str : "+str);
            String str1=obj.toString();
            System.out.print("\n str1 :"+str1);
      }
}
```

Bin>javac tostring1.java
Bin>java tostring1

Bank :name=kumar accno=105
str : Bank Details : name=kumar accno=105
str1 : name=kumar accno=105

Explanation

In the above program, the statement **bank obj=new bank();** creates an object to bank class and whose reference is assigned to obj as shown in above memory. The statement **obj.setbank("kumar",105);** calls the **setbank(String n, int a)** method of bank class, which assigns **"kumar"** to **name** and **105** to **accno.**

In the statement **System.out.print("\nBank :"+obj);** as the **obj** is used in the expression(print method) therefore **toString()** method of bank class is called and it return "**name=kumar accno=105**" to the **print()** method which prints the message

Bank :name=kumar accno=105

In the statement **String str="Bank Details :" +obj;** as **obj** is used again in the expression therefore **toString()** method is called which returns "**name=kumar accno=105**" and concatenates with the string **"Bank Details :"** and stores into **str** string object. The next print statement prints **str : name=kumar accno=105.**

In the statement **String str1=obj.toString(); obj** is calling the **toString()** method explicitly, which is a valid call. The **toString()** returns the string which assigns to **str1** and the next print statement prints **str1**; **name=kumar aceno=105.**

Creation of Custom (or) User-Defined Exceptions

Java contains built-in exceptions that handle most common errors, even though there some situations where we want to create our own exceptions according to the application requirements. Java allows programmers to create their own exceptions which are called as user-defined exceptions. The user-defined exception class should satisfy the following conditions to behave like an exception class.

1. User-defined exception class should extend from **Exception** class.
2. User-defined exception class should contain **toString()** method to return the error message.

Note: User-defined exception classes are checked execeptions therefore they should be listed with throws keyword.

The **Exception** class of **java.lang** package does not contain any methods of its own, it inherits methods from its superclass **Throwable.** Therefore all the sub-classes (Pre-defined or User-defined) have methods defined by **Thowable** available to them. The methods of **Throwable** class are shown in the following table.

Methods	Meaning
Throwable **fillInStackTrace**()	Fills in the execution stack trace.
Throwable **getCause**()	Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String **getLocalizedMessage**()	Creates a localized description of this throwable.
String **getMessage**()	Returns the detail message string of this throwable.
StackTraceElement[]**getStackTrace**()	Provides programmatic access to the stack trace information printed by printStackTrace().
Throwable **initCause**(Throwable cause)	Initializes the *cause* of this throwable to the specified value.

void **printStackTrace**()	Prints this throwable and its backtrace to the standard error stream.
void **printStackTrace**(PrintStream s)	Prints this throwable and its backtrace to the specified print stream.
void **printStackTrace**(PrintWriter s)	Prints this throwable and its backtrace to the specified print writer.
void **setStackTrace**(StackTraceElement[] stackTrace)	Sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.
String **toString**()	Returns a short description of this throwable.

A program to demonstrate creation of user-defined exception

We write a program to create a user-defined exception class to handle an exception while creation of bank account. The minimum deposit to create an account is Rs.1000

Bin>edit userexcep.java

```
class Deposit Exception extends Exception
{
  private String errmsg="";
  public  DepositException (String msg)
  {
        errmsg=msg;
  }
  public String toString()
  {
        return errmsg;
  }
}
class bank
{
  private String name;
  private int accno;
  private double bal;
  public void createaccount(String n, int ano,
                double amt)throws DepositException
  {
```

```
        try
        {
            if(amt<1000)
               throw new DepositException("Minimum deposit
                                  is 1000");
            name=n;
            accno=ano;
            bal=amt;
            System.out.print("\n Account created.");
        }
        catch(DepositException e)
        {
              System.out.print("\nError : "+e);
        }
    }
    public void showaccount()
    {
      System.out.print("\nname=" +name+"\t accno="+accno+
                                          "\t bal="+bal);
    }
}

  class userexcep
  {
    public static void main(String argv[])
                              throws DepositException
    {
          bank obj1=new bank();
          obj1.createaccount("kumar",105,5000.00);
          System.out.print("\nobj1...");
          obj1.showaccount();
          bank obj2=new bank();
          obj2.createaccount("raju",102,900.00);
          System.out.print("\n obj2...");
          obj2.showaccount();
    }
  }
```

```
Bin>javac userexcep.java
Bin>java userexcep
  Account created.
  obj1...
  name=kumar  accno=105  bal=5000.00
  Error : Minimum deposit is 1000
  obj2...
  name=null  accno=0  bal=0.0
```

Explanation:

When the above program is executed, the statement **bank obj1=new bank();** creates an object of **bank** class and assigns to **obj1** as shown in above memory. The statement **obj1.createaccount("kumar",105,5000.00);** calls **createaccount(String n,int ano,double amt)** and **"kumar"** passes to **n**, **105** to **ano**, **5000.00** to **amt**. In the method, the condition **if(bal<1000)** gets false and values of **n**, **ano**, **amt** assigns to **name**, **accno** and **bal**. The statement **obj1.showaccount()** prints the details of the object as

name=kumar accno=105 bal=5000.00

The statement **bank obj2=new bank();** creates an object of **bank** class and assigns to **obj2** as shown in above memory. The statement **obj2.createaccount("raju",102,900.00);** calls **createaccount(String n,int ano,double amt)** and **"raju"** passes to **n**, **102** to **ano**, **900.00** to **amt**. In the method, the condition **if(bal<1000)** gets true and the statement **throw new DepositException("Minimum deposit is 1000");** creates an object of **DepositException** class as a result it calls the constructor **DepositException(String msg)** and passes **"Minimum deposit is 1000"** to **msg** which then assigns to **errmsg**. The **throw** keyword throws the **DepositException** object and it will be caught by the catch block which prints the message **Error : Minimum deposit is 1000** and method terminates as a result the object(**obj2**) is not initializes with values.

The statement, **obj2.showaccount();** prints the **obj2** details as

name=null accno=0 bal=0.0

The **obj2** is not stored with values because of exception.