Chapter 10

Exception Handling in Java

Exception Handling

Exception Handling is the mechanism to handle runtime malfunctions. We need to handle such exceptions to prevent abrupt termination of program. The term exception means exceptional condition, it is a problem that may arise during the execution of program. A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

Exception

A Java Exception is an object that describes the exception that occurs in a program. When an exceptional events occurs in java, an exception is said to be thrown. The code that's responsible for doing something about the exception is called an **exception handler**.

Dictionary Meaning: Exception is an abnormal condition. In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling

Exception Handling is a mechanism to handle runtime errors.

Advantage of Exception Handling

The core advantage of exception handling is that normal flow of the application is maintained.

It makes for clear, robust, fault-tolerant programs.

Exception normally disrupts the normal flow of the application that is

why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statements will be executed. That is why we use exception handling.

Some common examples of Exception are:

- Divide by zero errors.
- Accessing the elements of an array beyond its range.
- Invalid input
- Opening a non-existent file.
- Heap memory exhausted.

Types of Errors

Errors may broadly classified into two categories:

1. Compile-time errors
2. Run-time errors

Compile Time Errors

All syntax errors will be detected and display by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler display an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

i. Most common compile time problems are:
ii. Missing semicolons.
iii. Missing (or mismatch of) brackets in classes and methods
iv. Misspelling of identifier and keywords
v. Use of undeclared variables
vi. Incompatible types in assignments.
vii. Use of = in place of == operator.

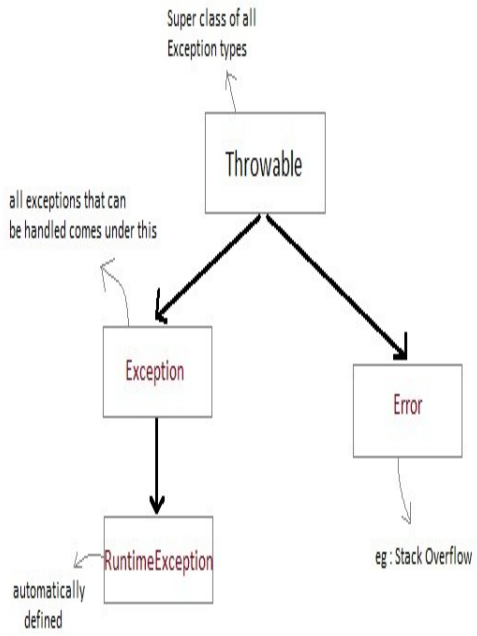
Run-Time Errors

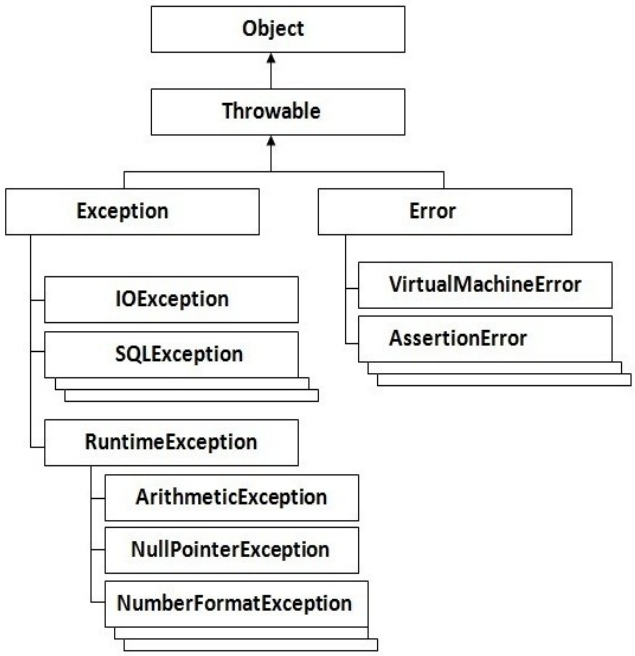
Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time error are:

i. Dividing an integer by zero
ii. Accessing an element that is out of the bounds of an array.
iii. Trying to store a value into an array of an incompatible class or type.
iv. Trying to illegally change the state of a thread.
v. Attempting to use negative size for an array.

Hierarchy of Exception classes

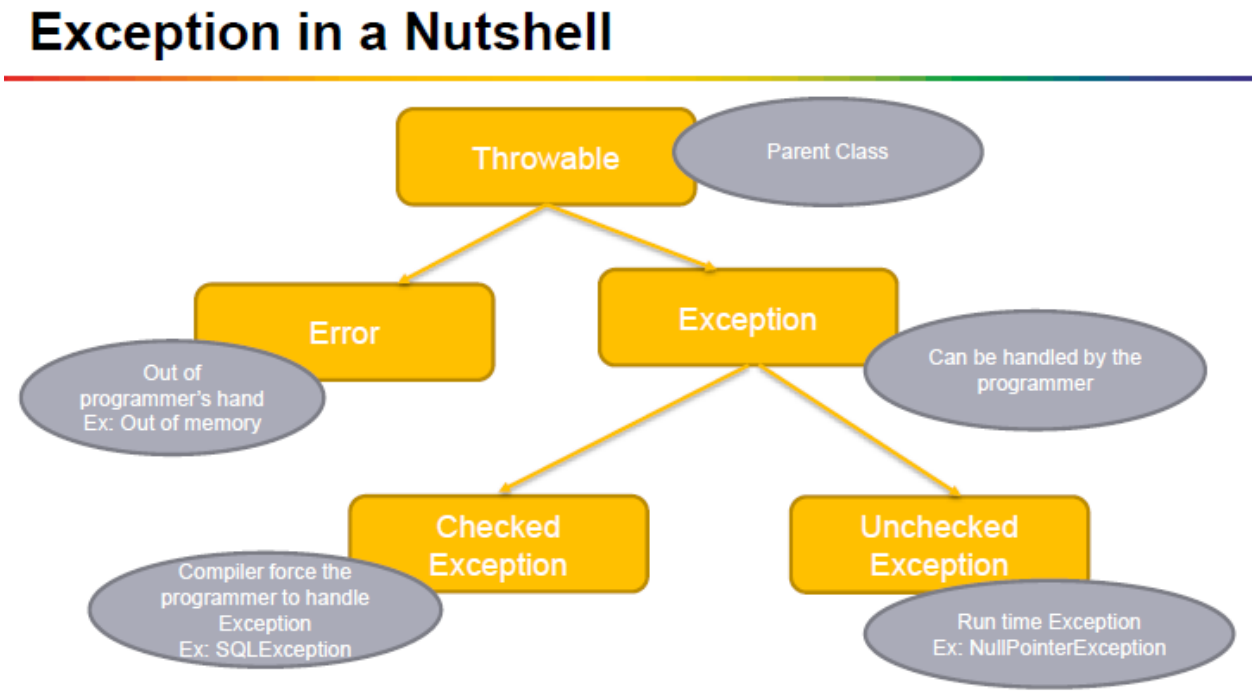
All exception types are subclasses of class **Throwable**, which is at the top of exception class hierarchy.





- **Exception** class is for exceptional conditions that program should catch. This class is extended to create user specific exception classes.
- **RuntimeException** is a subclass of Exception. Exceptions under this class are automatically defined for programs.

Exception are categorized into 3 category.



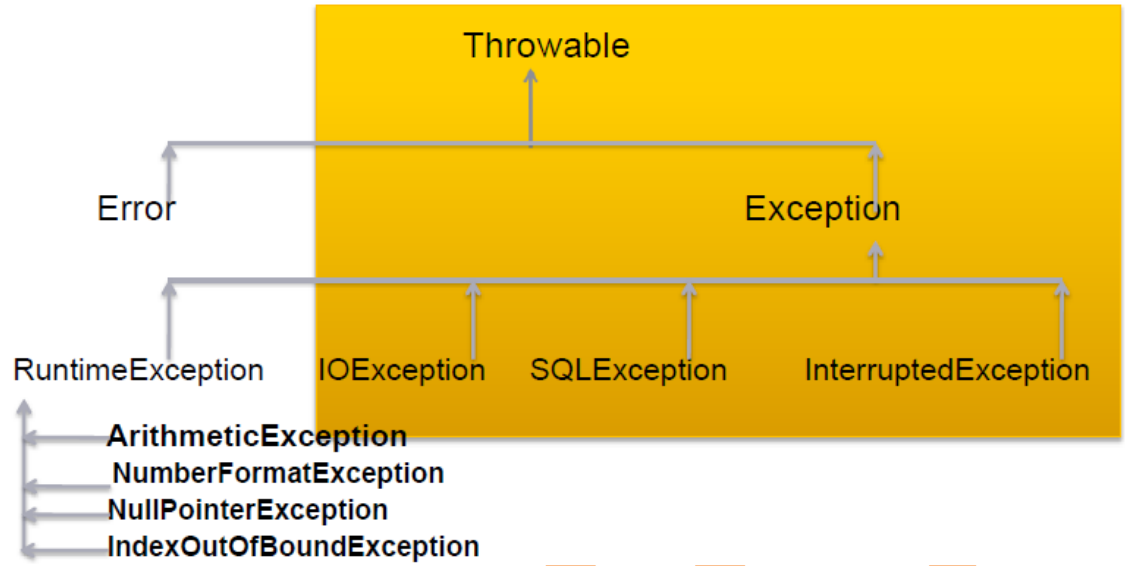
- **Checked Exception**

  Compiler force the programmer to handle Exception. E.g IOException, SQLException

A checked exception is an exception that usually happens due to user error or it is an error situation that cannot be foreseen by the programmer. A checked exception must be handled using a try or catch or at least declared to be thrown using throws clause. Non compliance of this rule results in a compilation error Ex: FileNotFoundException If you try to open a file using

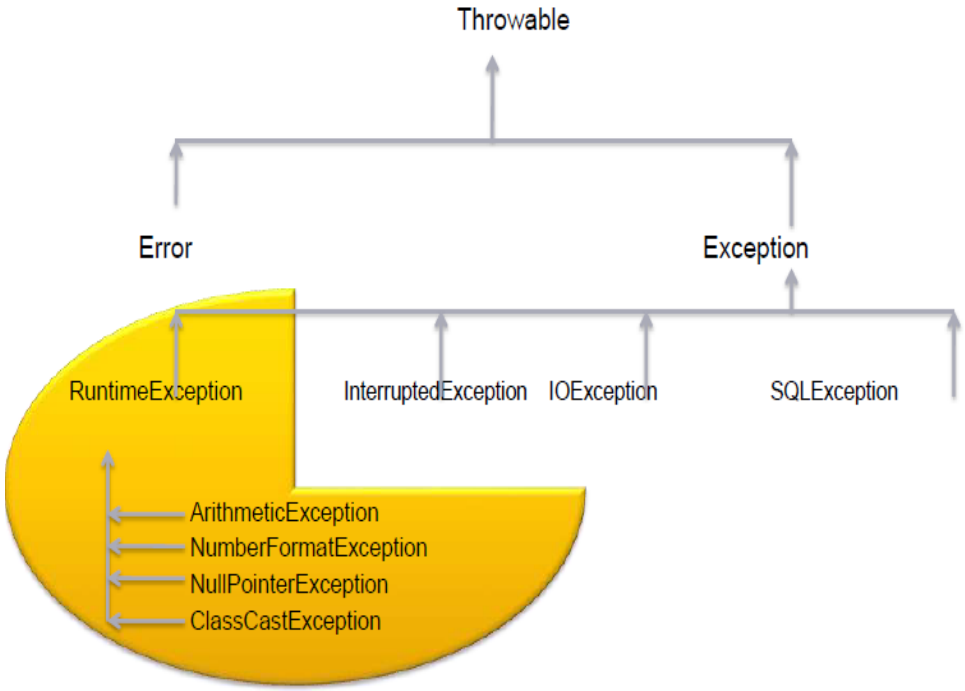
FileInputStream fx = new FileInputStream("A1.txt");
During execution, the system will throw a FileNotFoundException, if the file A1.txt is not located, which may be beyond the control of a

programmer.



- **Unchecked Exception**

  Unchecked exceptions are the class that extends **RuntimeException**. Unchecked exception are ignored at compile time. *Example :* ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.



Errors

Errors are typically ignored in code because you can rarely do anything about an error.(Out of programmer hand)

Example : if *stack overflow* occurs, an error will arise. This type of error is not possible handle in code.

Common scenarios of Exception Handling where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

5) Scenarios of StackOverflowError Occurs

```
public class Tester {
public static void recursivePrint(int num)
{
System.out.println("Number: " + num);
if(num == 0)
return;
else
recursivePrint(++num);
}
public static void main(String[] args) {
recursivePrint(1);
}
}
```

Exception in thread "main" java.lang.StackOverflowError

Handling Exception

Five keywords used in Exception handling:

1. try
2. catch
3. finally
4. throw
5. throws

Java uses the try-catch-finally syntax to test (i.e., try) a section of code and if an error occurs in that region, to trap (i.e., catch) the error. Any number of catches can be set up for various *exception* types. The **cacth {}** block is also responsible for handling the exceptions. The finally keyword can be used to provide a block of code that is always performed regardless of whether an exception is occurred or not.

try block

Enclose the code that might throw an exception in try block. It must be used within the method and must be followed by either catch or finally block.

Syntax of try with catch block

```
try
{
...
}
catch(Exception_class_Name reference)
{
...
}
```

Syntax of try with finally block

```
try
{
...
}
finally
{
...
}
```

Syntax of try with catch & finally block

```
try
{
...
}
catch(Exception_class_Name reference)
{
...
}
finally
{
...
}
```

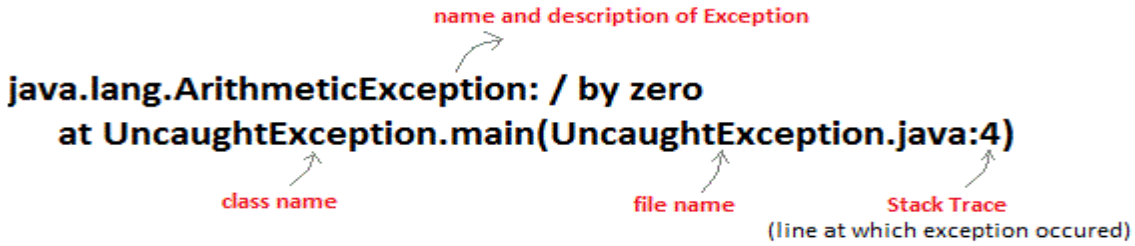
Program that generate exception but without error-handling

```
class Simple
{
   public static void main(String args[])
   {
       int data=50/0;
      System.out.println("rest of the code...");
   }
}
```

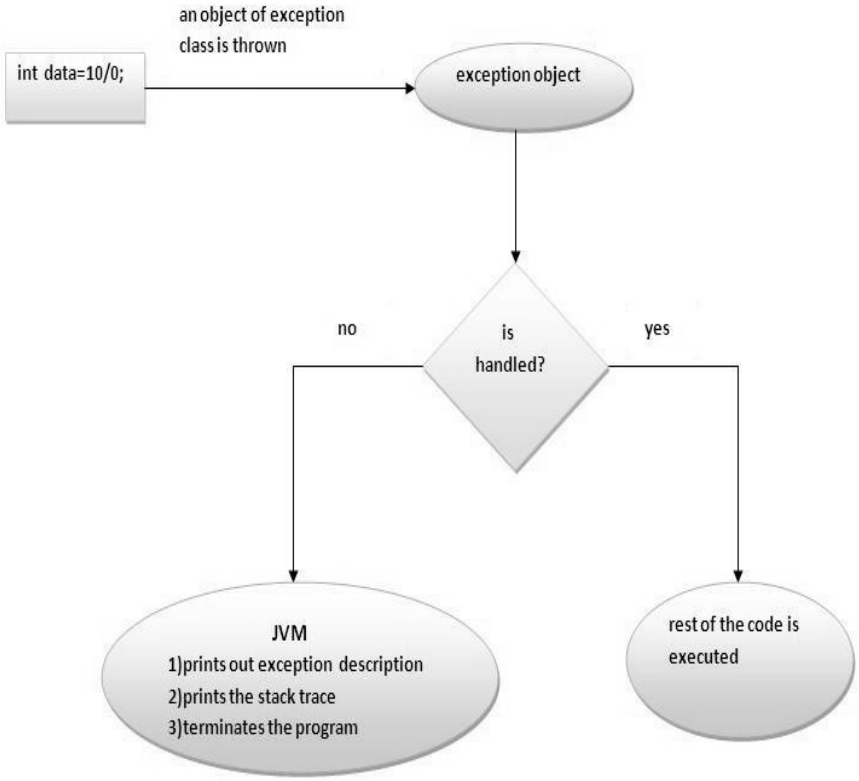
As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed. Let's see what happens behind the scene:

This will lead to an exception at runtime, hence the Java run-time system will construct an exception and then throw it. As we don't

have any mechanism for handling exception in the above program, hence the default handler will handle the exception and will print the details of the exception on the terminal.



What happens behind the code int a=50/0;



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 exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Program that generate exception with error-handling

```
class Simple
{
  public static void main(String args[])
  {
     try
     {
         int data=50/0;

     }
     catch(ArithmeticException e)
     {
          System.out.println(e);
       }
        System.out.println("rest of the code...");
   }
}
```

Output: **Exception in thread main java.lang.ArithmeticException:/ by zero**
rest of the code...

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

Multiple catch block:

If you have to perform different tasks at the occurrence of different Exceptions, use multiple catch block.

```
//Example of multiple catch block
 class Excep4
 {
     public static void main(String args[])
     {
         try
         {
             int a[]=new int[5];
             a[5]=30/0;
            String s = null;
           s.length();
          int n = Integer.parseInt(“Hello”);
          }
         catch(ArithmeticException e)
         {
         System.out.println("Division by zero not possible");
         }

         catch(ArrayIndexOutOfBoundsException e)
         {
         System.out.println("task 2 completed");
         }
         catch(Exception e)
        {
           System.out.println("common task completed");
```

```
    }


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

Output: **task1 completed**
rest of the code...

Rule: At a time only one Exception is occurred and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for **ArithmeticException** must come before catch for Exception .

```
class Excep4
{
   public static void main(String args[])
   {
       try
       {
            int a[]=new int[5];
            a[5]=30/0;
       }
       catch(Exception e)  {
System.out.println("common task completed"); }
       catch(ArithmeticException e)  {
System.out.println("task1 is completed");  }
```

```
      catch(ArrayIndexOutOfBoundsException e)  {
System.out.println("task 2 completed");  }

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

Output: **Compile-time error**

Nested try block:

try block within a try block is known as nested try block.

Why use nested try block?

try statement can be **nested** inside another block of **try**. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case if inner **try** block does not have a **catch** handler for a particular exception then the outer **try** is checked for match.

Syntax:

```
....
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
```

```
   catch(Child Class Exception e)
   {
   }
}
catch(Exception e)
{
}
....
```

Example:

```
//Example of nested try block

class Excep6
{
    public static void main(String args[])
    {
          try
         {
           try
           {
                  System.out.println("going to divide");
                   int b =39/0;
            }
               catch(ArithmeticException e)
            {
                System.out.println(e);
            }
             try
            {
            int a[]=new int[5];
                a[5]=4;
             }
```

```
            catch(ArrayIndexOutOfBoundsException e)
            {
           System.out.println(e);
            }

            System.out.println("other statement);
        }
      catch(Exception e)
      {
          System.out.println("handeled");
      }

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

Example for Unreachable Catch block

While using multiple **catch** statements, it is important to remember that exception sub classes inside **catch** must come before any of their super classes otherwise it will lead to compile time error.

```
class Excep
{
 public static void main(String[] args)
 {
  try
  {
   int arr[]={1,2};
   arr[2]=3/0;
  }
  catch(Exception e)     //This block handles all
Exception
```

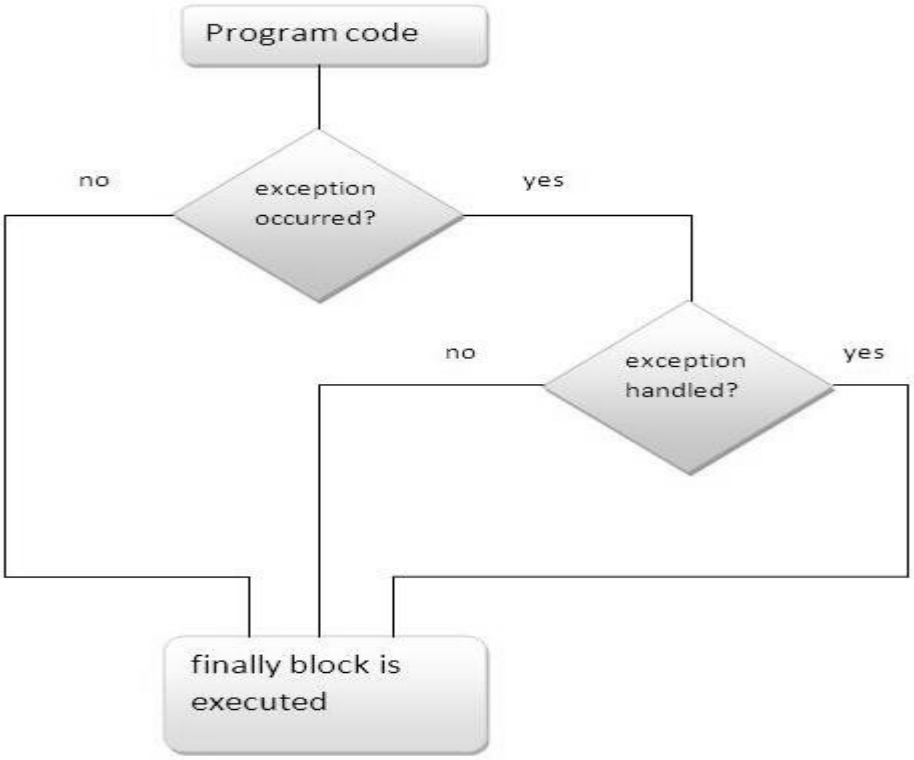
```
   {
    System.out.println("Generic exception");
   }
   catch(ArrayIndexOutOfBoundsException e)
//This block is unreachable
   {
    System.out.println("array index out of bound
exception");
   }
  }
}
```

Important points to Remember

1. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
2. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
3. While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.
4. In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.
5. Only the object of Throwable class or its subclasses can be thrown.

finally block

The finally block is a block that is always executed. It is mainly used to perform some important tasks such as closing connection, stream etc.



Note: Before terminating the program, JVM executes finally block(if any).

Note: finally must be followed by try or catch block.

Why use finally block?

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

case 1

Program in case exception does not occur

```
class Simple
{
   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: **5**
finally block is always executed
rest of the code...

case 2

Program in case exception occurred but not handled

```
class Simple
{
    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

Program in case exception occured and handled

```
class Simple{
 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...

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling **System.exit()** or by causing a fatal error that causes the process to abort).

throw keyword (Forcing an Exception)

The *throw* keyword is used to force an exception or the throw keyword is used to explicitly throw an exception. That means, you as programmer can force an exception to occur through **throw** keyword. The throw keyword is mainly used to throw custom exception and it can also pass a custom message to your exception handling module. We can throw either checked or uncheked exception.

Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

A throw statement causes the current method to immediately stop executing, much like return statement, and the exception is thrown to the previous method on the call stack. For example , the following statement

throw a new *ArrayIndexOutOfBoundException* with 5 being the invalid index:

```
throw new ArrayIndexOutOfBoundsException(5);
```

You can also instantiate (i.e. create) an exception object and throw it in a separate statement:

```
ArrayIndexOutOfBoundsException e = new ArrayIndexOutOfBoundsException(5);
throw e;
```

or

```
try
{
   Exception e1 = new Exception("User Defined Exception");
   ....
    Throw e1;
}
catch(Exception e)
 {
    System.out.println("ERROR: " + e.getMessage( ) );
}
```

You can throw only object of type *java.lang.Throwable*. In almost all situation, you will throw an object that is a child of *java.lang.Exception.* Recall that Exception class extends the *throwable* class.

Example of throw keyword

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 Excep13{

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

  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
  }
}
```

Output: **Exception in thread main java.lang.ArithmeticException:not valid**

throws Keyword

If a method invokes code that cause an exception, then the method should provide a catch clause to handle the exception. If a catch does not handle the exception, then the exception must be passed out of the method and be handled by the code calling the method. If an exception is allowed to be passed through method, a **throws** keyword is required in the method declaration to indicate that an exception can occur that the method itself does not handle. It is specified with method header, e.g.

public static void main(String args[]) throws Exception → indicating that if an exception occur

it will automatically report it to

error handler or JVM

Advantage : No Checked Exception can be propagated (forwarded in call stack).

Program which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Simple
{
   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[])
```

```
  {
      Simple obj=new Simple();
      obj.p();
      System.out.println("normal flow...");
   }
}
```

Output:**exception handled**
normal flow...

Exception propagation:

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: **By default Unchecked (*`RuntimeException`, `Error`, and their subclasses`)` Exceptions are forwarded in calling chain (propagated).**

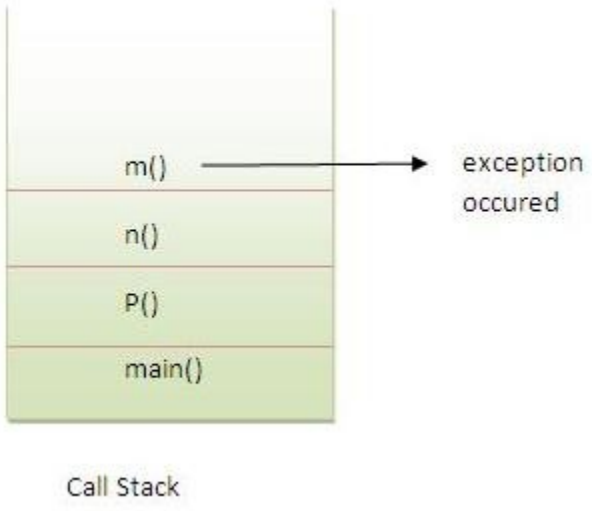
Program of Exception Propagation

```
class Simple
{
   void m()
   {
      int data=50/0;
   }
    void n()
    {
        m();
```

```
  }
  void p()
{
    try
    {
      n();
    }
    catch(Exception e)
    {
       System.out.println("exception handled");
     }
  }

  public static void main(String args[])
  {
      Simple obj=new Simple();
      obj.p();
      System.out.println("normal flow...");
   }
}
```

Output: **exception handled**
normal flow...



In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

```
class Simple
{
        void m() throws IOException
        {
        IOException err = new IoException("Device Error");
          throw err;
        }
          void n() throws IOException
        {
```

```
        m();
    }
    void p()
    {
        try
        {
            n();
        }
        catch(Exception e)
        {System.out.println("exception handeled");}
    }

     public static void main(String args[])
    {
        Simple obj=new Simple();
        obj.p();
        System.out.println("normal flow");
    }
}
```

Output:Compile Time Error

Custom Exception

Though Java provide an extensive set of in-built exceptions, there are cases in which we may need to define our own exception in order to handle the various specific errors that we might encounter.

While defining a user defined exception, we need to take care of the following aspects:

- The user defined exception class should extend from *Exception* class.

- The **toString( )** method may be overridden in the user defined exception class in order to display meaningful information about the exception.

Example of User Define Exception by extending Exception class.

```
class InvalidAgeException extends Exception
{
    InvalidAgeException(String s)
    {
        super(s);
    }
}
class Excep13
 {

     static void validate(int age)
throws InvalidAgeException
     {
       if(age<18)
```

```
            throw new InvalidAgeException("not valid");
        else
          System.out.println("welcome to vote");
    }

    public static void main(String args[])
    {
        try
        {
            validate(13);
        }
        catch(Exception m)
        {

System.out.println("Exception occured: "+m);
        }

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

Output: Exception occured: InvalidAgeException:not valid
rest of the code...

Example of user define exception handling using overridden toString() Method

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

 class NegativeAgeException extends
Exception
{
    private int age;
    NegativeAgeException (int age)
    {
        this.age = age;
    }
    public String toString()
    {
        return "Age cannot be negative" + "
" + age;

    }
}

class CustomeExceptionTest2
{

    static int getAge()
    {
        return -10;
    }
```

```
    public static void main(String[] args)
throws Exception
    {
        // TODO Auto-generated method stub
        int age = getAge();
    //  InputStreamReader r=new
InputStreamReader(System.in);
        //BufferedReader br=new
BufferedReader(r);

        //System.out.println("Enter Your Age
");
        //age =
Integer.parseInt(br.readLine());

        if(age < 0)
        {
            throw new
NegativeAgeException(age);

        }
        else
        {
            System.out.println("Age entered
is " + age);
        }
```

```
    }
}
```

Output : Exception in thread "main" Age cannot be negative -10
    at CustomeExceptionTest2.main(CustomeExceptionTest2.java:37)

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1. import java.io.*;
2. class M{
3.  void method()throws IOException
4.  {
5.     throw new IOException("device error");
6.  }
7. }
8.
9.   class Test
10.         {
11.           public static void main(String args[])
12.           {
13.              try
14.              {
15.                 Test t=new Test();
16.                  t.method();
17.               }
18.              catch(Exception e)
19.              {
20.                 System.out.println("exception handled");
```

```
21.                 }
22.
23.                 System.out.println("normal flow...");
24.             }
25.         }
```

Output:exception handled
normal flow...

Case2: You declare the exception

- A) In case you declare the exception, if exception does not occur, the code will be executed fine.
- B) In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1. import java.io.*;
2. class M
3. {
4.     void method()throws IOException
5.     {
6.       System.out.println("device operation performed");
7.     }
8. }
9.
10.
11.       class Test
12.       {
13.           public static void main(String args[])throws IOException
14.           {//declare exception
15.               Test t=new Test();
16.                t.method();
```

```
17.                 System.out.println("normal flow...");
18.             }
19.         }
```

Output:device operation performed
normal flow...

B)Program if exception occurs

```
1. import java.io.*;
2. class M
3.  {
4.      void method() throws IOException
5.      {
6.          throw new IOException("device error");
7.      }
8.  }
9.
10.
11.      class Test
12.      {
13.        public static void main(String args[])throws IOException
14.        {//declare exception
15.            Test t=new Test();
16.             t.method();
17.
18.            System.out.println("normal flow...");
19.         }
20.      }
```

Output:Runtime Exception

Difference between throw and throws:

throw keyword	throws keyword
1)throw is used to explicitly throw an exception.	throws is used to declare an exception.
2)checked exception can not be propagated without throws.	checked exception can be propagated with throws.
3)throw is followed by an instance.	throws is followed by class.
4)throw is used within the method.	throws is used with the method signature.
5)You cannot throw multiple exception	You can declare multiple exception e.g. public void method()throws IOException,SQLException.

ExceptionHandling with MethodOverriding

There are many rules if we talk about method overriding with exception handling. The Rules are as follows:

- **If the superclass method does not declare an exception**
  - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
  - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or

no exception but cannot declare parent exception.

If the superclass method does not declare an exception

1) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class Child extends Parent{
7.   void msg()throws IOException{
8.     System.out.println("child");
9.   }
10.         public static void main(String args[]){
11.          Parent p=new Child();
12.          p.msg();
13.         }
14.       }
```

Output:Compile Time Error

---

2) Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

```
1. import java.io.*;
2. class Parent{
3.   void msg(){System.out.println("parent");}
4. }
5.
6. class Child extends Parent{
```

```
7.   void msg()throws ArithmeticException{
8.     System.out.println("child");
9.   }
10.          public static void main(String args[]){
11.           Parent p=new Child();
12.           p.msg();
13.          }
14.        }
```

Output:child

If the superclass method declares an exception

1) Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

```
1. import java.io.*;
2. class Parent{
3.   void msg()throws ArithmeticException{System.out.println("pare
   nt");}
4. }
5.
6. class Child extends Parent{
7.   void msg()throws Exception{System.out.println("child");}
8.
9.   public static void main(String args[]){
10.          Parent p=new Child();
11.          try{
12.          p.msg();
13.          }catch(Exception e){}
```

```
14.         }
15.       }
```

Output:Compile Time Error

Example in case subclass overridden method declares same exception

```
1. import java.io.*;
2. class Parent
3. {
4.     void msg() throws Exception
5.     {
6.         System.out.println("parent");
7.     }
8. }
9. class Child extends Parent
10.       {
11.          void msg() throws Exception
12.          {
13.              System.out.println("child");
14.          }
15.          public static void main(String args[])
16.        {
17.           Parent p=new Child();
18.           try
19.           {
20.             p.msg();
21.           } catch(Exception e){}
22.         }
23.       }
```

Output:child

Example in case subclass overridden method declares subclass exception

```
1. import java.io.*;
2. class Parent{
3.   void msg()throws Exception{System.out.println("parent");}
4. }
5.
6. class Child extends Parent{
7.   void msg()
   throws ArithmeticException{System.out.println("child");}
8.
9.   public static void main(String args[]){
10.          Parent p=new Child();
11.          try{
12.          p.msg();
13.          }catch(Exception e){}
14.         }
15.       }
```

Output:child

Example in case subclass overridden method declares no exception

```
1. import java.io.*;
2. class Parent
3. {
4.     void msg()throws Exception
5.     {System.out.println("parent");}
6. }
7.
8. class Child extends Parent
9. {
10.          void msg()
```

```
11.          {System.out.println("child");}
12.
13.         public static void main(String args[])
14.        {
15.          Parent p=new Child();
16.          try
17.          {
18.          p.msg();
19.          }catch(Exception e){}
20.         }
21.        }
```

Output:child