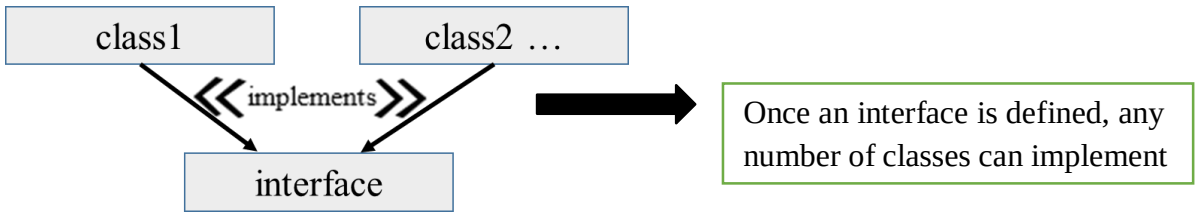
Chapter-3

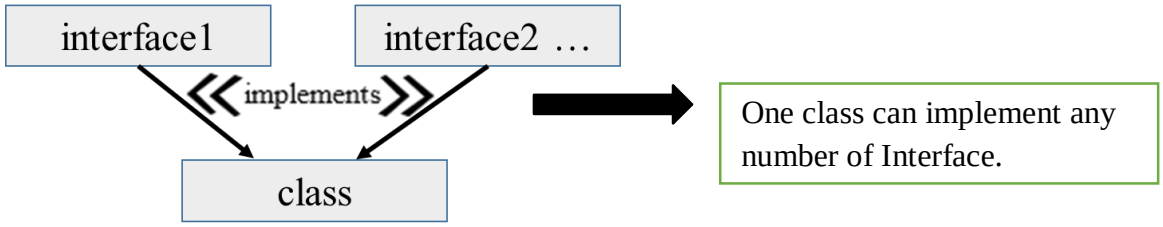
Interfaces

Interface Fundamentals:

- An Interface defines a set of variables and methods that will be implemented by a class.
- is not a class but similar to abstract classes, except that no method can include a body.
- An interface contains the behaviour that class must implement.
- An interface is a collection of abstract methods and final variables.
-



-



Creating an Interface:

An interface is defined by use of the ***"interface"*** keyword.

Syntax:

```
access-specifier interface name {
      // any no. of final, static fields
      // any no. of abstract method declarations
   ret-type method-name1(param-list);
      ……
   ret-type method-nameN(param-list);
}
```

- Each method in an interface is implicitly abstract, so the **abstract** keyword is not needed.
- Methods in an interface are implicitly **public**.

Example:

```
interface Stack {
      void push(int item);// default public and abstract
      int pop();       // default public and abstract
      int MAX = 3;    // default public static and abstract
}
```

Implementing an Interface:

- One or more class can implement interface.
- Class that implement interface must provide definition to all abstract methods of that interface.

Two ways of implementing interface.

1. class *implements* interface
2. class *extends* class and *implements* interface

1. class *implements* interface:

Syntax:

```
class ClassName implements interfaceName{
     //implement abstract methods of interface
     // other stuff
}
```

2. class *extends* class and *implements* interface :

Syntax:

```
class ClassName extends SuperClass implements interfaceName{
     //implement abstract methods of interface
     // other stuff
}
```

Example:

```
interface Shape{
     double area(double length, double breadth);
     static final String color = "Red";
 }
class Rectangle implements Shape  {
     public double area (double length, double breadth ) {
          return(length * breadth);
     }
}
class InterfaceDemo{
     public static void main(String[] a){
          Rectangle rect = new Rectangle();
          double areaRect = rect.area(2.0,4.0) ;
          System.out.println( "Area = " + areaRect );
          System.out.println( "Color = " + rect.color );
     }
}
```

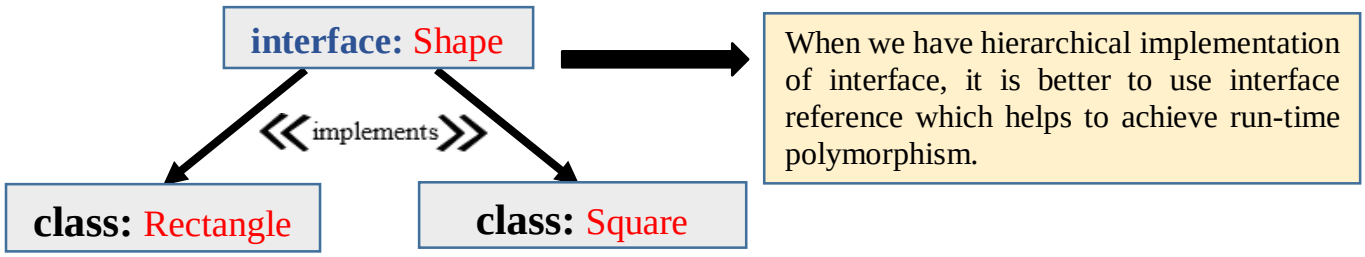
Output:

```
E:\JAVAExamples>java InterfaceDemo
Area = 8.0
Color = Red
```

Using Interface References:

- Interface cannot be instantiated. But we can declare a reference of an interface type.
- Interface references are used to refer to objects of types that implement the interface.

Example: Consider the following structure



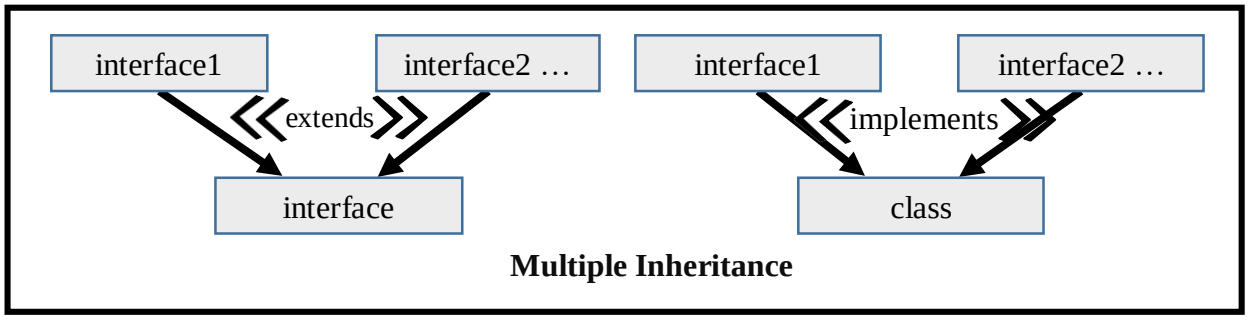

```
interface Shape{
      void getArea( );
}
class Rectangle implements Shape{
      double height, width;
      Rectangle (double height, double width){
            this.height = height;
            this.width = width;
      }
      public void getArea() {  // must be public
            System.out.println ("Area of Rectangle= "+(height*width));
      }
}
class Square implements Shape {
      double side;
      Square (double side) {
            this.side = side;
      }
      public void getArea() {     // must be public
            System.out.println ("Area of Square= " + (side*side));
      }
}
class InterfaceDemo2 {
      public static void main(String [] a){
            Shape sh;
            sh = new Rectangle(20.00, 40.00);
            sh.getArea();
            sh = new Square(15.5);
            sh.getArea();
      }
}
```

Output:

```
E:\JAVAExamples>java InterfaceDemo2
Area of Rectangle = 800.0
Area of Square = 240.25
```

Implementing Multiple Interfaces:

Multiple inheritance can be achieved only through the interfaces.



Two ways of implementing Multiple inheritance:

i. class *implements* Multiple interfaces

ii. interface *extends* Multiple interfaces or interfaces can be **extended**

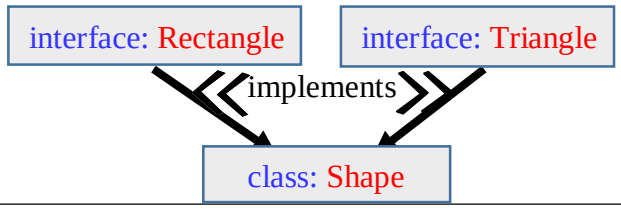
i. class *implements* Multiple interfaces:

A class can implement more than one interface at a time.

Syntax:

```
class ClassName implements interface1,interface2{
      //code
}
```

Example: **Lab Programs -** Write Simple Program on Java for the implementation of Multiple inheritance using interfaces to calculate the area of a rectangle and triangle




```
interface Rectangle{
      public void areaRectangle (double width, double height);
}
interface Triangle  {
      public void areaTriangle (double base, double height);
}
class Shape implements Rectangle, Triangle {
      public void areaRectangle (double width , double height ) {
            double area =  width * height ;
            System.out.println ("The Area of Rectangle is " + area);
      }
      public void areaTriangle (double base, double height) {
            double area = (0.5 * base * height);
            System.out.println("The Area of Triangle is " + area);
      }
}
```

```
Continued…
public class MultiInheritance {
     public static void main (String args[]) {
          Shape sh = new Shape();
          sh.areaRectangle(10.00, 5.00);
          sh.areaTriangle(8.5, 4.5);
     }
}
```

Output:

```
E:\JAVAExamples>java MultiInheritance
The Area of Rectangle is 50.0
The Area of Triangle is 19.125
```

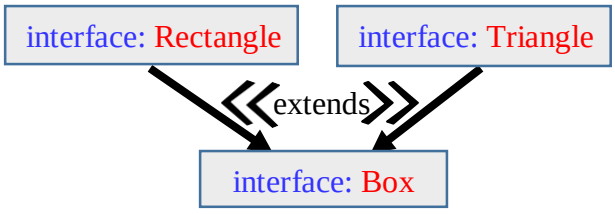
ii. interface *extends* Multiple interfaces or **interfaces** can be **extended :**

An interface can extend more than one interface at a time.

Syntax:

```
interface InterfaceName extends interface1,interface2{
     //code
}
```

Example:




```
interface Rectangle{
     public void areaRectangle(double width, double height);
}
interface Triangle{
     public void areaTriangle(double base, double height);
}
interface Box extends Rectangle, Triangle{
     public void showName( String name);
}

class Shape implements Box{
     public void areaRectangle(double width, double height){
      System.out.println ("The Area of Rectangle is" + width*height);
     }
     public void areaTriangle(double base, double height){
          System.out.println("The Area of Triangle is " +
                                                  (0.5*base*height));
     }
```

```
Continued….
     public void showName(String name){
          System.out.println("Box= " + name);
     }
}
class MulInterface{
     public static void main(String[] args){
          Shape sh = new Shape();
          sh.showName("Rectangle");
          sh.areaRectangle(10.00, 5.00);
          sh.showName("Triangle");
          sh.areaTriangle(8.5, 4.5);

     }
}
```

Output:

```
E:\JAVAExamples>java MulInterface
Box= Rectangle
The Area of Rectangle is 50.0
Box= Triangle
The Area of Triangle is 19.125
```

Constants in Interfaces:

- An interface can also include "variables" that are not instance variable.
- All variables declared inside interface is ***implicitly public***, ***static*** & ***final.***
- Interface can be used create constants that will have fixed value in all classes that implement it.

Example:

```
interface IConst{
     int MIN= 0;//implicitly all variables public static final
     int MAX = 10;
     String ERROR = "Boundary Error";
}
class ConstDemo implements IConst{
    public static void main(String[] args){
        int[] nums = new int[MAX];
        for(int i=MIN; i <= MAX; i++){
            if(i >= MAX)
                System.out.println(ERROR);
            else{
                nums[i] = i;
                System.out.print(nums[i] + " ");
               }
          }
     }
}
```

Output:

```
E:\JAVAExamples>java ConstDemo
0 1 2 3 4 5 6 7 8 9 Boundary Error
```

Nested Interfaces:

- An interface can be declared a member of another interface or of a class. Such an interface is called **member interface** or **a nested interface**.
- Nested interfaces are used to group related interfaces. So, Easy to maintain.
- The nested interface must be referred by the outer interface or class. It *can't be accessed* directly.
- Nested interfaces are implicitly static and public.

Two ways we can defined Nested interface they are:

i. Nested interface within the interface:

```
interface OuterInterface{
    ...  ...
      interface  NestedInterface{
      ...
     }
}
```

ii. Nested interface within the class:

```
class className{
    ...  ...
      interface NestedInterface{
      ...
     }
}
```

Example:

```
interface Showable{
      void show();
      interface Message{
            void msg();
      }
}

class NestedInterface implements Showable.Message, Showable{
      public void msg(){
            System.out.println("Hello nested interface");
      }
      public void show(){
            System.out.println("Hi from Outer Interface");
      }
```

```
Continued….

public static void main(String args[]){
          Showable.Message message = new NestedInterface();
          message.msg();

          Showable sh=new NestedInterface();
          sh.show();
     }
}
```

Output:

```
E:\JAVAExamples>java NestedInterface
Hello nested interface
Hi from Outer Interface
```

NOTE:

- When the abstract method of interface is not implemented (implementation not provided) in the sub-class, compiler will generate an error.
- If subclass method tries to modify the value of variable declared in interface, the compiler will generate error.

Example:

```
interface Shape {
     double area ( );
     String color = "Red";
}
class Rectangle implements Shape {
     Rectangle (String color ) {
          super.color = "BLUE" ; //color is constant generates error
     }
     // No implementation of area() //generates error
}
```

Final thoughts on interface:

Differences between class and interface:

class	interface
Can instantiate a class.	Can't instantiate an interface.
Can have any number of constructors	Does not contain any constructor
All methods in interface must be abstract	Contain only non-abstract methods.
Contain any type of variables	Contains only static and final variables.
Sub-class extends class	Sub-class implements interface
An interface cannot extend multiple inheritance	An interface can implement multiple inheritance
The **class** keyword is used to declare abstract class.	The **interface** keyword is used to declare interface.

Differences between abstract class and interface:

Abstract class	interface
Can have abstract and non-abstract methods.	Can have only abstract methods.
Doesn't support multiple inheritance.	Supports multiple inheritance.
Can have final, non-final, static and non-static variables.	Has only static and final variables.
Can have static methods, main method.	Can't have static methods, main method or constructor.
Can provide the implementation of interface.	Can't provide the implementation of abstract class.
The **abstract** keyword is used to declare abstract class.	The **interface** keyword is used to declare interface.

Chapter-2

Package

Package Fundamentals:

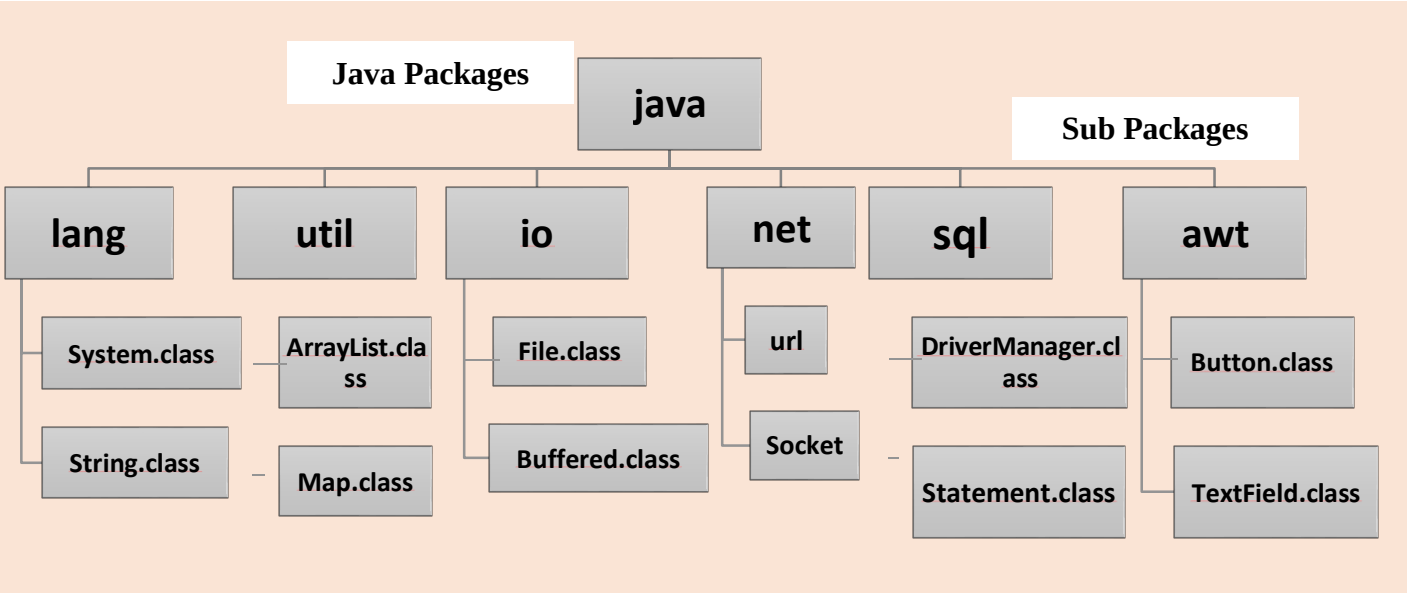
A java package is a collections of classes, interfaces, sub-packages, Exceptions etc.,

In java, packages can be categorized in two form, built-in package and user-defined package.

- **Built-in Packages**
- **User-Defined Packages:** Packages are created by user to categorized classes and interfaces.

Note: All classes in Java belong to ***some package.*** When no package has been explicitly specified, the ***default*** *or* ***global*** package is used.

Built-in Packages: Packages are pre-defined such as java, lang, util, io, awt, net, sql, javax, swings



User-Defined Packages: Packages are created by user to categorized classes and interfaces.

How to create packages?

To create a package, you will use the **"package"** keyword.

Syntax: **package pkg_name;** Here, **pkg_name** is the name of the package.

- More than one file can include the same **package** statement.
- You can create **hierarchy** of packages.

Syntax: **package pack1.pack2.pack3…..packN;**

Example: package rnsit.pg.mca;

Example:

```
package mypack;

public class Simple{
     public static void main(String args[]){
          System.out.println("Welcome to package");
     }
}
```

How to compile and run?

There are two ways to compile and run the package programs.

Method-1: Creating package in the **current directory**

To Compile: \> javac –d • directory filename.java

To Run: Give the class name path which is having main method.

\> java packagename.filename

```
E:\JAVAExamples>javac -d . Simple.java

E:\JAVAExamples>java mypack.Simple
Welcome to package
```

Method-2: Creating package in the **different directory.**

To Compile:

\> javac –d directoryname:\ filename.java

Set Classpath:

\> set CLASSPATH=;.; directoryname:\ ;.;

To Run:

\> java packagename.filename

```
E:\JAVAExamples>javac -d C:\ Simple.java

E:\JAVAExamples>set CLASSPATH=;.; C:\ ;.;

E:\JAVAExamples>java mypack.Simple
Welcome to package
```

Note:

- The **'-d'** is a switch, that tells the compiler to create a given package.
- The **(.) operator** represents the current folder.

Advantages:

- **Avoid name collisions.**
  - ✓ With thousands of Java libraries available, it's inevitable that class and interface names will be duplicated.
  - ✓ Packages allow you to use two different classes or interfaces with the same name.
- **Encapsulate more than one class.**
  - ✓ A set of classes might need access to each other, but still be hidden from the outside world.
  - ✓ to categorize the classes and interfaces so that they can be easily maintained

Packages and Member Access:

- The visibility of an element is determined by its access specification- **private, public, protected**, or **default** and the package in which it resides.
- If a class has default access, it can be accessed only by other code within its same package.

	Private member	Default member	Protected member	Public member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

Importing Packages:

Using import, you can bring one or more members of a package into view.

Syntax: < **import *pkg.classname*;** >

Where,

pkg is the name of the package, which can include its full path

classname is the name of the class being imported.

There are three ways to access the package from outside the package.

- **import package.*;** Example: import pack1.*;
- **import package.classname;** Example: import pack1.A;
- **fully qualified name** Example: pack1.A obj=new pack1.A();

Note: If you import a package, subpackages will not be imported.

Example: Lab Program-7 - **Complete the following:**

i. **Create a package named shape.**
ii. **Create some classes in the package shape like Square, Triangle, & Circle.**
iii. **Import and compile these classes in other program.**

Square.java

```
package Shape;
public class Square{
     private double side;
     public void setSquare(double s){
          side=s;
     }
```

Continued…

```
public class Square{
     private double side;
     public void setSquare(double s){
          side=s;
     }
     public void area(){
          double area=side*side;
          System.out.println("Area of Square=" +area);
     }
}
```

Triangle.java

```
package Shape;

public class Triangle{
     private double height, breadth;
     public void setTriangle(double hg, double br){
          height=hg;
          breadth=br;
     }
     public void area(){
          double area=0.5*height*breadth;
          System.out.println("Area of Triangle= " +area);
     }
}
```

Circle.java

```
package Shape;
public class Circle{
     private double radius;
     public void setCircle(double rad){
          radius=rad;
     }
     public void area(){
          double area=0.5*3.14*radius*radius;
          System.out.println("Area of Circle=" +area);
     }
}
```

```
import  Shape.Triangle;
import Shape.Square;
import  Shape.Circle;

public class Lab7 {
     public static void main (String [] args) {
          Triangle rect = new Triangle();
          rect.setTriangle(5.6, 6.4);
          rect.area();

          Square sq = new Square();
          sq.setSquare(10.5);
          sq.area();

          Circle round = new Circle();
          round.setCircle(5.6);
          round.area();
     }
}
```

How to compile and run?

There are two ways to compile and run the package programs.

Method-1: Creating package in the **current directory**

To Compile: \> javac –d • directory filename.java

To Run: Give the class name path which is having main method.

\> java packagename.filename

```
E:\JAVAExamples\LabPrograms>javac -d . Square.java

E:\JAVAExamples\LabPrograms>javac -d . Circle.java

E:\JAVAExamples\LabPrograms>javac -d . Triangle.java

E:\JAVAExamples\LabPrograms>javac Lab7.java

E:\JAVAExamples\LabPrograms>java Lab7
Area of Triangle= 17.919999999999998
Area of Square=110.25
Area of Circle=49.2352
```

Method-2: Creating package in the **different directory.**

To Compile: \> javac –d directoryname:\ filename.java

Set Classpath: **\> set CLASSPATH=;.; directoryname:\ ;.;**

To Run: Give the class name path which is having main method.

\> java packagename.filename

```
E:\JAVAExamples\LabPrograms>javac -d C:\ Triangle.java

E:\JAVAExamples\LabPrograms>javac -d C:\ Circle.java

E:\JAVAExamples\LabPrograms>javac -d C:\ Square.java

E:\JAVAExamples\LabPrograms>set CLASSPATH=;.; C:\ ;.;

E:\JAVAExamples\LabPrograms>javac Lab7.java

E:\JAVAExamples\LabPrograms>java Lab7
Area of Triangle= 17.919999999999998
Area of Square=110.25
Area of Circle=49.2352
```

Importing java's Standard Packages:

Java defines a large number of standard classes that are available to all programs.

Subpackage	Description
java.lang	Contains a large number of general-purpose classes
java.io	Contains the I/O classes
java.net	Contains those classes that support networking
java.Applet	Contains classes for creating applets
java.awt	Contains classes that support the Abstract Window Toolkit
java.util	Contains various utility classes, plus the Collections Framework.

Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly.

- There is no need to qualify it by the class name.
- Static imports are used to save your time and typing.
- If you hate to type same thing again and again then you may find such imports interesting.

There are two general forms of the import static statement.

i. **import static pkg-typename.staticmembername;**

ii. **import static pkgtypename.*;**

Example:

```
class StaImport{
    public static void main(String args[]){
          double var1= Math.sqrt(5.0);
          double var2= Math.tan(30);
          System.out.println("Square of 5 is:"+ var1);
          System.out.println("Tan of 30 is:"+ var2);
     }
}
```

The above program uses **sqrt()**, **tan()** are the static methods from java-built-in class **Math**, which is part of **java.lang.** Because they must be called through their class's name.

Furthermore, having to specify the class name each time **sqrt()** or **tan()** are used become tedious.

You can eliminate the tedium of specifying the class name through the use of ***static import*** and Of course, static import is not limited just to the Math class or just to those methods that as shown in the following example:

```
import static java.lang.System.*;
import static java.lang.Math.*;

class StaImport1{
    public static void main(String args[]){
          double var1= sqrt(5.0);
          double var2= tan(30);
          out.println("Square of 5 is:"+ var1);
          out.println("Tan of 30 is:"+ var2);
     }
}
```

Output:

```
E:\JAVAExamples>java StaImport1
Square of 5 is:2.23606797749979
Tan of 30 is:-6.405331196646276
```

Chapter-3

Exception Handling

Introduction:

Bug: An error in a program is called **bug.**

Error: Error is something that most of the time you cannot handle it.

Types of error in software

- **Syntax error** **:** Due to the fact that the syntax of the language is not respected.
- **Semantic error** **:** Due to the fact to an improper use of program statements.
- **Logical Errors** **:** Due to the fact that the specification is not respected.

When errors are detected?

- **Compile Time errors:** Syntax and semantic errors indicated by the compiler.
- **Runtime errors** **:** Dynamic semantic errors and logical errors that cannot be detected by the compiler (debugging).

What is Debugging?

Finding the error from a program is called debugging.

The Exception Hierarchy:

What is Exception?

An **exception** is a runtime event which interrupts the normal flow of the program. When an exception occurs, program processing gets terminated and doesn't continue further.

Exception is a class in java

Exception Hierarchy

- All exception classes are derived from a class called **java.lang.Exception**, but Exception is a subclass of the **Throwable** class.
- So **Throwable** class is the *superclass* of all **errors** and **exceptions** in the Java language.
- When an exception occurs in a program, an ***object*** of some type of exception class is generated.

There are mainly three types of exceptions:

i. Checked Exception

ii. Unchecked Exception

iii. Error

Definitions:

i. **Checked Exception:**

A checked exception is an exception that occurs at the compile time, these are also called as **compile time exceptions**. These exceptions cannot be ignored at the time of compilation, the programmer should handle these exceptions. Example: IOException, SQLException etc.

ii. **Unchecked Exception:**

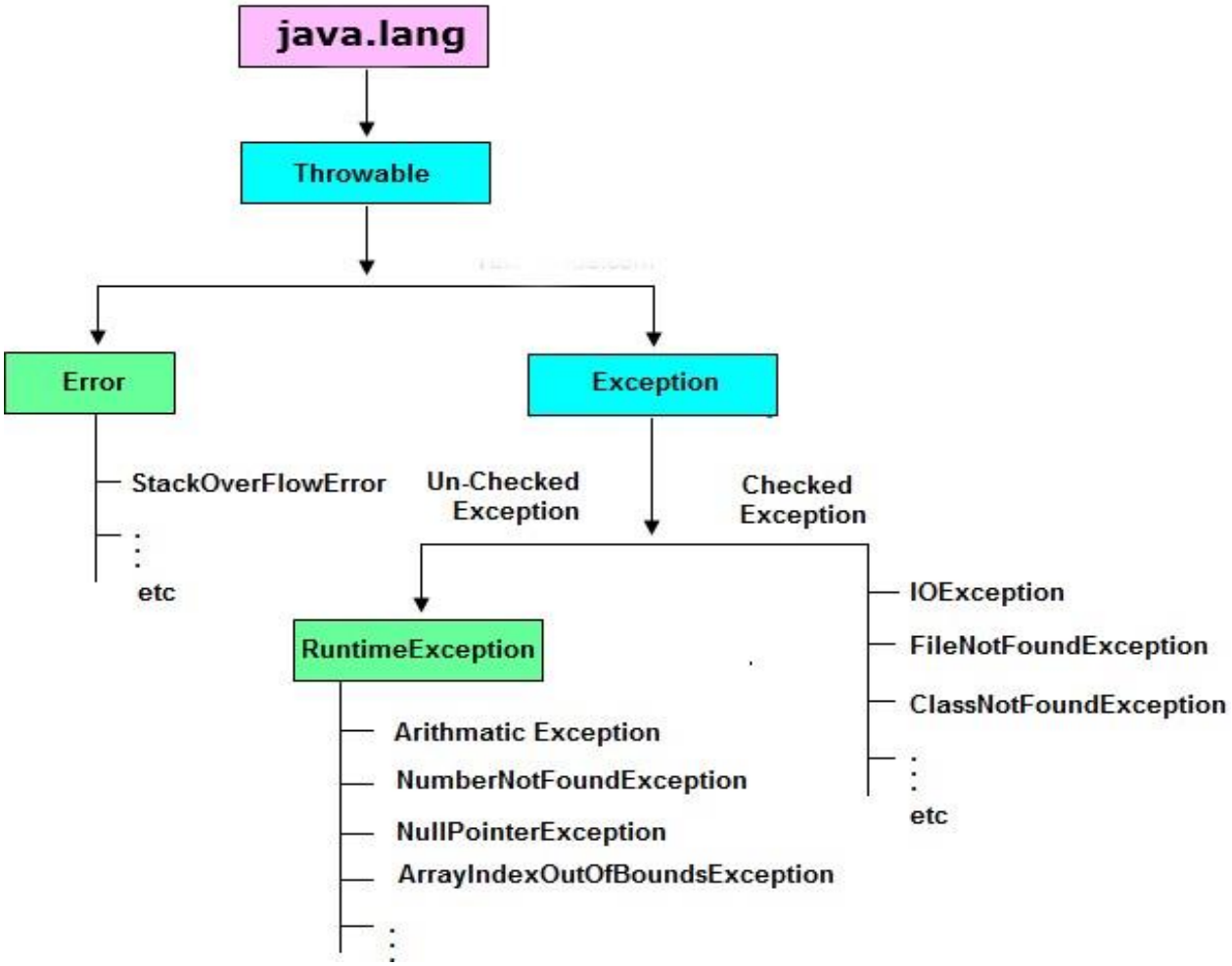
An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

iii. **Error:**

Error is **irrecoverable** or can explain these are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error.

Example: OutOfMemoryError, VirtualMachineError, etc. these are also ignored at the time of compilation.



Exception Handling Fundamentals:

Java exception handling is managed via 5 keywords. They are

- **try** : Programs statements that you want to monitor for exceptions
- **catch** : To handle the Exception.
- **throw** : To manually throw an exception
- **throws** : An exception that is *thrown* out of a method
- **finally** : To *execute important code* while exiting such as *closing* connection, stream etc.

try {- -}:

- is used to enclose the code that might throw an exception. It must be used within the method.
- Java try block must be followed by **either catch or finally block**.

Syntax of **try{}-catch{}:**

```
try {
      //code that may throw exception
}
catch(Exception_class_Name ref){
      //code
}
```

Syntax of **try{}-finally{}:**

```
try {
      //code that may throw exception
}
finally{
      //code
}
```

catch {- -}:

- Java catch block is used to handle the Exception. It must be used after the try block only.
- You can use multiple catch block with a single try.

Example: Let's try to understand the problem if we don't use try-catch block.

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

Output:

```
E:\JAVAExamples>java TestTC
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestTC.main(TestTC.java:5)
```

In the above example, when the expression **50/0** is executing it throw the exception hence that rest of the code (In such case rest of the code... statement is not printed) is not executed. If there are 100 lines of code after exception, so all the code after exception will not be executed. Because those exception is handled by JVM standard Exception hence it terminates once the runtime error occurred. Solution for this is use **try** and **catch** block as shown in the below program.

```
class TestTC{
     public static void main(String args[]){
          try{
               int data=50/0; //may throw exception
          }
          catch(ArithmeticException e)
          {
               System.out.println("Error is "+ e);
          }
          System.out.println("rest of the code...");
     }
}
```

Output:

```
E:\JAVAExamples>java TestTC
Error is java.lang.ArithmeticException: / by zero
rest of the code...
```

Now, in the above example, rest of the code is executed i.e. rest of the code... statement is printed. Because the expression **50/0** written within try block hence whenever the exception thrown that catch it in the catch block.

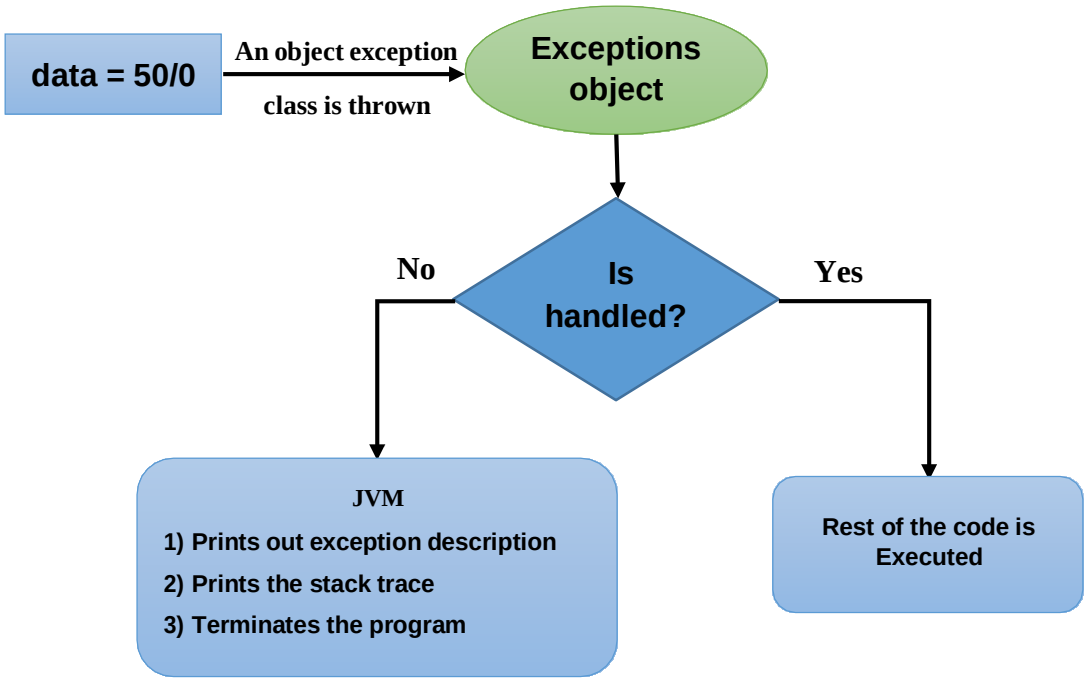
The Consequences of an Uncaught Exception:

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. This internal working is represented in the below flowchart for the given example.

Internal working of java try-catch block



It is important to understand that the type of the exception must match the type of specified in a catch block. If it does not, the exception would not be caught. Then such exception is again handled by JVM.

Example:

```
class TestTC{
     public static void main(String args[]){
          try{
               int data=50/0; //may throw exception
          }
          catch(ArrayIndexOutOfBoundsException e)
          {
               System.out.println("Error is "+ e);
          }
          System.out.println("rest of the code...");
     }
}
```

In this program tries to catch an ArithmeticException with a catch for an **ArrayIndexOutOfBoundsException**, but it won't be caught by catch because it fails to overrun an **ArithmeticException**. Hence JVM will handled.

Note: **Stack trace include the following:**

- the method name, main;
- the filename; and
- the line number, 4
- also include the type of exception thrown is a subclass of Exception called **ArithmeticException**, which more specifically describes what type of error happened.

Exceptions Enable you to handle errors gracefully:

One of the key benefits of exception handling is that it enables your program to respond to an error in a graceful, rational way. In some cases, it may be possible to fix the problem and allow the program to continue running.

Example: open a file, network connection, database connection and other task.

```
class TestTCG
{
     public static void main(String args[]){
          int[] numer={4,8,16,32,64,128};
          int[] denom={2,0,4,4,0,8};

          for(int i=0;i<numer.length;i++)
          try{
               System.out.println(numer[i] + "/" + denom[i] +
                              "=" + numer[i] / denom[i]);
          }
          catch(ArithmeticException e){
               System.out.println("Error is: " +e);
          }
          System.out.println("rest of the code...");
     }
}
```

Output:

```
E:\JAVAExamples>java TestTCG
4/2=2
Error is: java.lang.ArithmeticException: / by zero
16/4=4
32/4=8
Error is: java.lang.ArithmeticException: / by zero
128/8=16
rest of the code...
```

Using Multiple catch clauses:

- If you have to perform different tasks at the occurrence of different Exceptions then you can associate more than one catch clause with a try.
- Each catch must catch a different type of exception.
- At a time only one Exception is occurred and at a time only one catch block is executed.

Syntax:

```
try {
     //code that may throw exception
}
catch(Exception_class_Name1 ref1){ //code }
catch(Exception_class_Name2 ref2){ //code }
```

```
class TestTCM
{
     public static void main(String args[]){
          int[] numer={4,8,16,32,64,128};
          int[] denom={2,0,4,4,0};

          for(int i=0;i<numer.length;i++)
          try{
               System.out.println(numer[i] + "/" + denom[i] +
                                  "=" + numer[i] / denom[i]);
          }
          catch(ArithmeticException e){
               System.out.println("Error is: " +e);
          }
          catch(ArrayIndexOutOfBoundsException e1){
               System.out.println("Error is: " +e1);
          }

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

Output:

```
E:\JAVAExamples>java TestTCM
4/2=2
Error is: java.lang.ArithmeticException: / by zero
16/4=4
32/4=8
Error is: java.lang.ArithmeticException: / by zero
Error is: java.lang.ArrayIndexOutOfBoundsException: 5
rest of the code...
```

As the output confirms, each catch responds only to its own type of exception.

In general, **catch** clauses are checked in the order in which they occur in a program. Only a matching clause is executed. All other **catch** blocks are ignored.

Catching subclass Exceptions:

- There is one important point about multiple catch clauses that relates to **subclasses.**
- A **catch** clause for a **superclass** will also match any of its **subclasses.**
- If you want to catch exceptions of both a **superclass** type and a **subclass** type, put the subclass **first** in the **catch** sequence.
- If you do not, then **superclass catch** will also catch all derived classes.

```
class TestTCMS{
     public static void main(String args[]){
          int[] numer={4,8,16,32,64,128,256,512};
          int[] denom={2,0,4,4,0};

          for(int i=0;i<numer.length;i++)
          try{
               System.out.println(numer[i] + "/" + denom[i] +
                                  "=" + numer[i] / denom[i]);
          }
          catch(ArithmeticException e){
               System.out.println("Error is: " +e);
          }
          catch(Exception e1){
               System.out.println("Error is: " +e1);
          }
          System.out.println("rest of the code...");
     }
}
```

Output:

```
E:\JAVAExamples>java TestTCMS
4/2=2
Error is: java.lang.ArithmeticException: / by zero
16/4=4
32/4=8
Error is: java.lang.ArithmeticException: / by zero
Error is: java.lang.ArrayIndexOutOfBoundsException: 5
Error is: java.lang.ArrayIndexOutOfBoundsException: 6
Error is: java.lang.ArrayIndexOutOfBoundsException: 7
rest of the code...
```

In this case, the first catch clause handles **ArithmeticException**. The second one catches all other program-related exceptions, including **ArrayIndexOutofBoundsException** generated when an index value of **denom** is 5, 6, 7 occurs.

The order of catch clauses in the preceding example is important. As an experiment, if you reverse the order of catch clauses like this

```
catch(Exception e){
     System.out.println("Error is: " +e);
}
catch(ArithmeticException e1){
     System.out.println("Error is: " +e1);
}
System.out.println("rest of the code...");
```

Although it **"looks right"**, this sequence won't compile. As explained, **a subclass exception must be caught before its superclass exception**. Hence, the 1st catch statement will catch all exceptions and the 2nd catch will never be reached, thus producing a compile-time error as show below.

```
E:\JAVAExamples>javac TestTCMS.java
TestTCMS.java:17: error: exception ArithmeticException has already been caught
                catch(ArithmeticException e1){
                ^
1 error
```

try blocks can be nested:

- One try block can be nested within another.
- An exception generated within the **inner try** block that is not caught by a **catch** associated with that **try** is propagated to the **outer try** block.

Why use nested try block?

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

Example: Here the **ArrayIndexOutOfBoundsException** is not caught by the **inner catch**, but by the **outer catch**.

```
class TestTCN {
     public static void main(String args[]){
          int[] numer={4,8,16,32,64,128,256,512};
          int[] denom={2,0,4,4,0,8};

          try{ //Outer try
               for(int i=0;i<numer.length;i++){
               try{ //Inner try
                    System.out.println(numer[i] + "/" + denom[i] +
                                        "=" + numer[i] / denom[i]);
                    }//Exit from Inner try
                    catch(ArithmeticException e){
                         System.out.println("Error is: " +e);
                    }
               }
             }//Exit from Outer try
               catch(ArrayIndexOutOfBoundsException e1){
                    System.out.println("Error is: " +e1);
               }
          System.out.println("rest of the code...");
     }
}
```

Output:

```
E:\JAVAExamples>java TestTCN
4/2=2
Error is: java.lang.ArithmeticException: / by zero
16/4=4
32/4=8
Error is: java.lang.ArithmeticException: / by zero
128/8=16
Error is: java.lang.ArrayIndexOutOfBoundsException: 6
rest of the code...
```

In this example, an exception that can be handled by the **inner try - -** here, a divide-by-zero error – – allows the program to continue. However, an array boundary error is caught by the outer try, here if it match the catch then allows the program to continue. Otherwise, which causes the program to terminate.

Throwing an Exception:

- **"throw"** keyword is used to explicitly throw an exception.
- We can throw either checked or unchecked exception
- It is mainly used to throw **custom exception**.

Syntax:

```
throw new Exceptionclass();
throw new Exceptionclass("Any Message");
```

Here, **Exceptionclass** is any type of an Exception class (such as custom Exception, checked and unchecked Exception) derived from Throwable.

Example:

```
class TestThrow{
     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[]){
          try{
          validate(13);
          }
          catch(ArithmeticException ex)
          {
               System.out.println("Error: " +ex);
          }
          System.out.println("rest of the code...");
     }
}
```

Output:

```
E:\JAVAExamples>java TestThrow
Error: java.lang.ArithmeticException: not valid
rest of the code...
```

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**.

Rethrowing an Exception:

- When you rethrow an exception, it will not be recaught by the same catch clause.
- It will **propagate** to an **outer catch**.
- To rethrow an exception use

Syntax:

```
throw exceptob;
```

Here, **exceptob** must be an object of an exception class derived from **Throwable**.

Example:

```
class TestReThrow {
     static void validate(int age){
          try{
               if(age<18)
                    throw new ArithmeticException("not valid");
               else
                    System.out.println("welcome to vote");
          }
          catch(ArithmeticException ex){
               System.out.println("Error is ");
               throw ex; //Rethrowing an exception
          }
     }

     public static void main(String args[]){
          try{
               validate(13);
          }
          //catch rethrown exception
          catch(ArithmeticException e){
               System.out.println(e);
          }
               System.out.println("rest of the code...");
     }
}
```

Output:

```
E:\JAVAExamples>java TestReThrow
Error is
java.lang.ArithmeticException: not valid
rest of the code...
```

In this program, **ArithmeticException** is handled locally, by **validate()**, but that ArithmeticException is **rethrown**. Here, that is caught by **main()**.

A Closer look at Throwable:

All exceptions are subclasses of **Throwable**, all exceptions support the methods defined by **Throwable**. The following is the list of important methods available in the Throwable class.

Method	Description
Throwable **fillInStackTrace()**	Returns a Throwable object that contains a completed stack trace
String **getLocalizedMessage()**	Returns a localized description of the exception
String **getMessage()**	Returns a description of the exception
void **printStackTrace()**	Displays the stack trace
void **printStackTrace(PrintStream Stream)**	Sends the stack trace to the specified stream.
void **printStackTrace(PrintWriter Stream)**	Sends the stack trace to the specified stream.
String **toString()**	Returns a String object containing a complete description of the exception.

Example:

```
class TestMethod {
     static void validate(int age){
          try{
               if(age<18)
                    throw new ArithmeticException("not valid");
               else
                    System.out.println("welcome to vote");
          }
          catch(ArithmeticException ex){
               System.out.println("Error is " +ex );
               System.out.println("\nStack Trace: " );
               ex.printStackTrace();
               System.out.println(ex.getMessage());
          }
     }
     public static void main(String args[]){
          validate(13);
     }
}
```

Output:

```
E:\JAVAExamples>java TestMethod
Error is java.lang.ArithmeticException: not valid

Stack Trace:
java.lang.ArithmeticException: not valid
        at TestMethod.validate(TestMethod.java:6)
        at TestMethod.main(TestMethod.java:20)
not valid
```

using finally:

- is used *to* ***execute important code*** such as closing connection, stream etc.
- is always executed whether **exception** is **handled** or **not**.
- must be followed by **try** or **catch** block.
- The finally clause is optional. However, each try block there can be **zero** or **more catch blocks**, but **only one finally block**.

Syntax:

```
try {
      //code that may throw exception
}
catch(Exception_class_Name1 ref1){ //code }
catch(Exception_class_Name2 ref2){ //code }
finally{
      //code
}
```

Example:

```
class TestTCF {
     public static void main(String args[]){
          try{
               int data=50/2; //may throw exception
          }
          catch(ArithmeticException e)
          {
               System.out.println("Error is "+ e);
          }
          finally{
               System.out.println("finally block is always executed");
          }
          System.out.println("rest of the code...");
     }
}
```

Output:

```
E:\JAVAExamples>java TestTCF
finally block is always executed
rest of the code...
```

Difference between final, finally and finalize:

final	finally	Finalize
➢ Final is used to apply restrictions on class, method and variable. ➢ Final class can't be inherited ➢ final method can't be overridden ➢ final variable value can't be changed.	➢ Finally is used to place important code ➢ it will be executed whether exception is handled or not.	➢ Finalize is used to perform clean up processing just before object is garbage collected.
"final" is a keyword.	**"finally"** is a block.	**"finalize"** is a method.

using throws:

Sometime method generates an exception that it does not handle, it must declare that exception in a **"throws"** clause. It is mainly used to handle the **checked** exceptions.

If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he/she is not performing checkup before the code being used.

Syntax:

```
Ret-type mehtName(param-list)throws Exce-list{
     //body
}
```

Here, **Exce-list** – is a common-separated list of checked exceptions that the method might throw outside of itself

Example:

```
import java.io.*;

class TestThrows1{
     public static void main(String args[])
                              throws IOException, Exception{
          int i;
          BufferedReader br= new BufferedReader(
                                   new InputStreamReader(System.in));
          try{
               System.out.println("Enter a number");
               i=Integer.parseInt(br.readLine());
```

```
Continued…
                if(i<10)
                     throw new ArithmeticException();
                else
                     System.out.println("i= " + i);
           }
           catch(IOException ex){
                System.out.println("Some Input error");
           }
           catch(Exception ex){
                System.out.println("Some Input error");
           }
      }
}
```

Output:

```
E:\JAVAExamples>java TestThrows1
Enter a number
12
i= 12
```

```
E:\JAVAExamples>java TestThrows1
Enter a number
r
Some Input error
```

Difference between throw and throws:

throw	throws
is used to explicitly throw an exception.	is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
is followed by an instance.	is followed by class.
is used within the method.	is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions. Example: public void method() throws IOException, SQLException.

Java's Built-in Exceptions:

- Inside the standard package java.lang , Java defines several exception classes.
- There are two types of exceptions:
  - **checked exceptions:** are checked at compile-time
  - **unchecked exceptions:** are checked at runtime.

Unchecked Exception:

Exception	Meaning
ArithmeticException	Arithmetic error, such as division 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 method
NullPointerException	Invalid use of a null reference
TypeNotPresentException	Type not found
NumberFormatException	Invalid conversion of a string to a numeric format
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string

Checked Exception:

Exception	Meaning
ClassNotFoundException	Class not found
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
ReflectiveOperationException	Superclass of reflection-related exceptions

New Exception features added by JDK 7:

There are 3 new features of Exception handling are added in JDK 7. They are:

i. try{ } expanded to **try-with-resources:** also referred as **automatic resource management**, is a new exception handling mechanism, which automatically closes the resources used within the try catch block.

   To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block.

Following is the **syntax** of **try-with-resources statement**.

Syntax:

```
try(FileReader fr = new FileReader("file path")) {
   // use the resource
   } catch () {
      // body of catch
   }
}
```

ii. Multi-catch: allows two or more exceptions to be caught by the same catch clause as show below.

Example:

```
try{
     System.out.println(numer[i] + "/" + denom[i] +
                              "=" + numer[i] / denom[i]);
   }
catch(ArithmeticException | ArrayIndexOutOfBoundsException e){
     System.out.println("Error is: " +e);
}
```

iii. Final rethrow or more precise rethrow:

Refer page No: 28

Creating Exception Subclasses:

- Creating an **own exception** is easy.
- Just define a subclass of **Exception** which is a subclass of **Throwable**.
- The Exception class does not define any methods of its own. It does inherit those methods provided by **Throwable**.
- You can override one or more of these methods in exception subclass that you create.
- Two commonly used Exception constructors are shown here:

```
Exception()
Exception(String msg);
```

Example:

```
class TestUDE{
     public static void main(String[] args){
          int i=5;
          try{
               if(i<10)
                    throw new MyException("Errorrr");
               else
                    System.out.println("I= " +i);
          }
          catch(Exception e){
               System.out.println("Message " +e);
          }
     }
}
//User defined Exeption
class MyException extends Exception{
     MyException(String msg){
          super(msg);
     }
}
```

Output:

```
E:\JAVAExamples>java TestUDE
Message MyException: Errorrr
```