JAVA PROGRAMMING

COURSE NAME: MCA – 5TH SEMESTER

COURSE CODE: MCA18501CR

Teacher Incharge: *Dr. Shifaa Basharat*
Contact: *fazilishifaa@gmail.com*

Inheritance:

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in java by which one class is allowed to inherit the features (fields and methods) of another class.

Basic terminology:

Super Class: The class whose features are inherited is known as super class (or a base class or a parent class).

Sub Class: The class that inherits the other class is known as sub class (or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is extends.

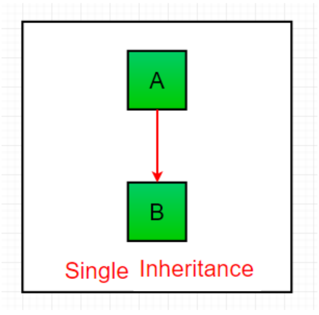
Syntax:

```
class derived-class extends base-class

{

  //methods and fields

}
```

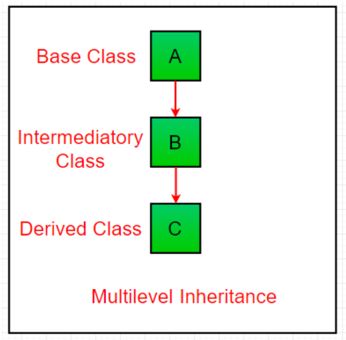
Types of Inheritance in Java

Below are the different types of inheritance which is supported by Java.

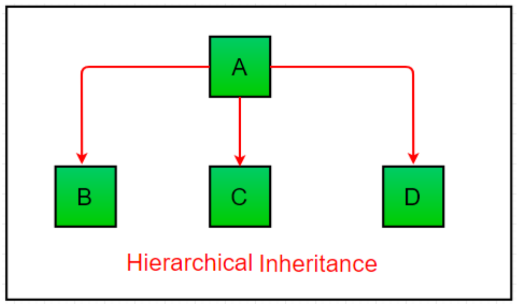
1. **Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.



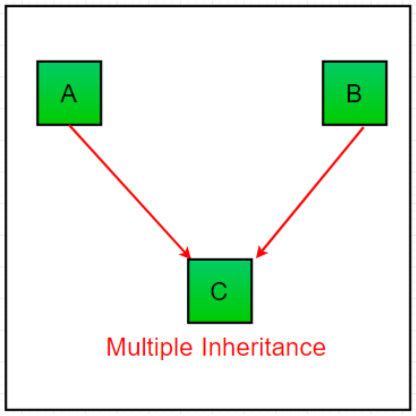
2. **Multilevel Inheritance:** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



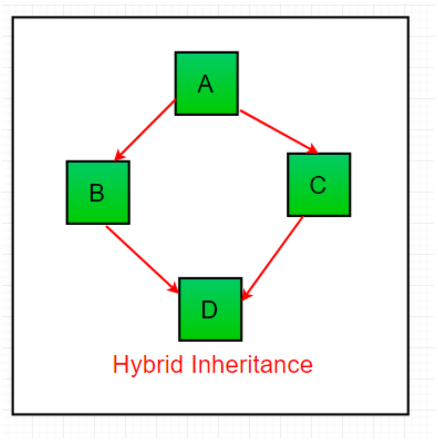
3. **Hierarchical Inheritance:** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B, C and D.



4. **Multiple Inheritance (Through Interfaces):** In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. However, Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



5. **Hybrid Inheritance (Through Interfaces):** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. Hence, we can achieve hybrid inheritance only through interfaces.



Important facts about inheritance in Java

- ***Default superclass:*** Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- ***Superclass can only be one:*** A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by java.
- ***Inheriting Constructors:*** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

- ***Private member inheritance:*** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

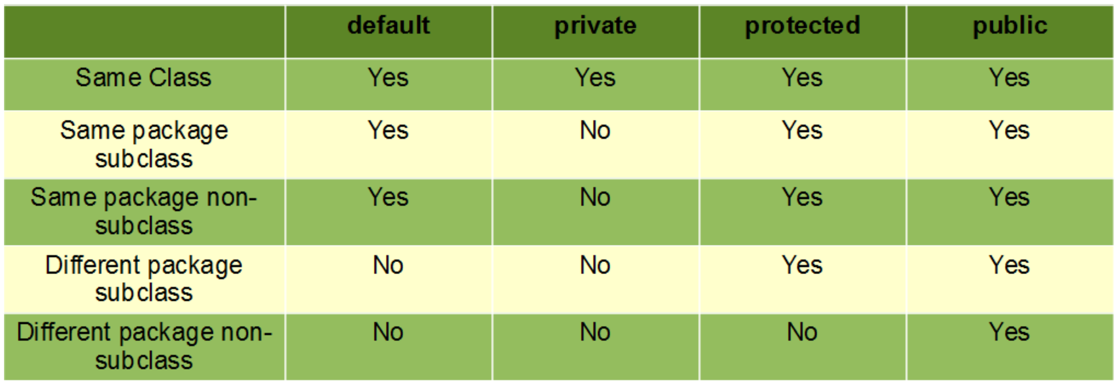
What all can be done in a Subclass?

- In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:
- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- We can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Role of Access Modifiers in inheritance:

Access modifiers in Java helps to restrict the scope of a class, constructor, variable, method or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- **Default**: When no access modifier is specified for a class, method or data member, it is said to be having the **default** access modifier by default. The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

- **Private**: The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared. Any other **class of same package will not be able to access** these members. Top level Classes or interface cannot be declared as private because private means "only visible within the enclosing class".

- **protected**: The protected access modifier is specified using the keyword **protected**. The methods or data members declared as protected are **accessible within same package or sub classes in different package.**

- **public**: The public access modifier is specified using the keyword **public**. The public access modifier has the **widest scope** among all other access modifiers. Classes, methods or data members which are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

1. Single Inheritance Example:

```
import java.util.*;
import java.lang.*;
import java.io.*;
public class Shape
{
   int length;
   int breadth;
}
/*****************************************************/
public class Rectangle extends Shape
{
   int area;
   public void calcualteArea()
   {
      area = length*breadth;
   }
   public static void main(String args[])
   {
      Rectangle r = new Rectangle();
      //Assigning values to Shape class attributes
      r.length = 10;
      r.breadth = 20;
      //Calculate the area
      r.calcualteArea();
      System.out.println("The Area of rectangle of length \""
            +r.length+"\" and breadth \""+r.breadth+"\" is \""+r.area+"\"");
   }
}
//Output
The Area of rectangle of length "10" and breadth "20" is "200"
```

2. Multilevel Inheritance Example:

```
import java.util.*;
import java.lang.*;
import java.io.*;
class Person
{
    private String name;
    Person(String s)
    {
          setName(s);
    }
    public void setName(String s)
    {
      name = s;
    }
    public String getName()
    {
      return name;
    }
    public void display()
    {
      System.out.println("Name of Person = " + name);
    }
}
/*********************************************************/
class Staff extends Person
{
    private int staffId;
    Staff(String name,int staffId)
    {
      super(name);
      setStaffId(staffId);
    }
    public int getStaffId() {
        return staffId;
    }
    public void setStaffId(int staffId) {
        this.staffId = staffId;
    }
    public void display()
    {
      super.display();
      System.out.println("Staff Id is  = " + staffId);
    }
}
/***************************************************************/
```

```
class TemporaryStaff extends Staff
{
    private int days;
    private int hoursWorked;
    TemporaryStaff(String sname,int id,int days,int hoursWorked)
    {
       super(sname,id);
       this.days  = days;
       this.hoursWorked = hoursWorked;
    }
    public int Salary()
    {
        int salary = days * hoursWorked * 50;

      return salary;
    }
    public void display()
    {
      super.display();
      System.out.println("No. of Days = " + days);
      System.out.println("No. of Hours Worked = " + hoursWorked);
      System.out.println("Total Salary = " + Salary());
    }
}
/*********************************************************/
public class MultilevelInheritanceExample
{
    public static void main(String args[])
    {
        TemporaryStaff ts = new TemporaryStaff("JavaInterviewPoint",999,10,8);
        ts.display();
    }
}
/**************************************************/
//Output
Name of Person = JavaInterviewPoint
Staff Id is  = 999
No. of Days = 10
No. of Hours Worked = 8
Total Salary = 4000
```

3. <u>Hierarchical Inheritance Example:</u>

```
import java.util.*;
import java.lang.*;
import java.io.*;

public class ClassA
{
   public void dispA()
   {
      System.out.println("disp() method of ClassA");
   }
}
/*************************************************************/
public class ClassB extends ClassA
{
   public void dispB()
   {
      System.out.println("disp() method of ClassB");
   }
}
/************************************************************/
public class ClassC extends ClassA
{
   public void dispC()
   {
      System.out.println("disp() method of ClassC");
   }
}
/*******************************************************************/
public class ClassD extends ClassA
{
   public void dispD()
   {
      System.out.println("disp() method of ClassD");
   }
}
/**************************************************************/
```

```
/*****************************************************************/
public class HierarchicalInheritanceTest
{
   public static void main(String args[])
   {
      //Assigning ClassB object to ClassB reference
      ClassB b = new ClassB();
      //call dispB() method of ClassB
      b.dispB();
      //call dispA() method of ClassA
      b.dispA();


      //Assigning ClassC object to ClassC reference
      ClassC c = new ClassC();
      //call dispC() method of ClassC
      c.dispC();
      //call dispA() method of ClassA
      c.dispA();

      //Assigning ClassD object to ClassD reference
      ClassD d = new ClassD();
      //call dispD() method of ClassD
      d.dispD();
      //call dispA() method of ClassA
      d.dispA();
   }
}
/********************************************************************/
//Output
disp() method of ClassB
disp() method of ClassA
disp() method of ClassC
disp() method of ClassA
disp() method of ClassD
disp() method of ClassA
```

4. Default Access Modifier Example:

```
//Java program to illustrate default modifier
package p1;

//Class defaultdemo1 is having Default access modifier
class defaultdemo1
{
   void display()
      {
         System.out.println("Hello World!");
      }
}
/*****************************************************************/
//Java program to illustrate error while using class from different package with
//default modifier
package p2;
import p1.*;

//This class is having default access modifier
class defaultdemonew
{
   public static void main(String args[])
      {
         //accessing class defaultdemo1 from package p1
         defaultdemo1 obj = new defaultdemo1();
           obj.display();
      }
}
/******************************************************************/
//Output
Compile Time Error
```

5. Private Access Modifier Example:

```
//Java program to illustrate error while using class from different package with private
 //modifier
package p1;
 class A
{
   private void display()
    {
        System.out.println("GeeksforGeeks");
    }
}
/*******************************************************************/
class B
{
   public static void main(String args[])
      {
          A obj = new A();
          //trying to access private method of another class
          obj.display();
      }
}
/*******************************************************************/
//Output
error: display() has private access in A
obj.display();
```

6. <u>Protected Access Modifier Example:</u>

```
//Java program to illustrate protected modifier
package p1;
//Class A
public class A
{
  protected void display()
   {
      System.out.println("Hello world");
   }
}
/*********************************************************/
//Java program to illustrate protected modifier
package p2;
import p1.*; //importing all classes in package p1

//Class B is subclass of A
class B extends A
{
  public static void main(String args[])
  {
     B obj = new B();
     obj.display();
  }
 }
/*************************************************************/
//Output
Hello world
```

7. Public Access Modifier Example:

```
//Java program to illustrate public modifier
package p1;
public class A
{
  public void display()
    {
       System.out.println("Hello world");
    }
}
/*************************************************************/
package p2;
import p1.*;
class B
{
   public static void main(String args[])
    {
       A obj = new A;
       obj.display();
    }
}
/*************************************************************/
//Output
Hello world
```

JAVA Super Keyword:

1. super() invokes the constructor of the parent class.
2. super.variable_name refers to the variable in the parent class.
3. super.method_name refers to the method of the parent class.

super() invokes the constructor of the parent class

super() will invoke the constructor of the parent class. Before getting into that we will go through the default behavior of the compiler. Even when you don't add **super()** keyword the compiler will add one and will invoke the **Parent Class constructor**.

Example 1:

```
class ParentClass
{
        public ParentClass()
        {
                System.out.println("Parent Class default Constructor");
        }
}
/**********************************************************************/
public class SubClass extends ParentClass
{
        public SubClass()
        {
                System.out.println("Child Class default Constructor");
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
        }
}
```

Output

```
Parent Class default Constructor
Child Class default Constructor
```

Even when we add explicitly also it behaves the same way as it did before.

```
class ParentClass
{
        public ParentClass()
        {
                System.out.println("Parent Class default Constructor");
        }
}
/**************************************************************************/
public class SubClass extends ParentClass
{
        public SubClass()
        {
                 super();
                System.out.println("Child Class default Constructor");
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
        }
}
```

Output

```
Parent Class default Constructor
Child Class default Constructor
```

You can also call the parameterized constructor of the Parent Class. For example, **super(10)** will call parameterized constructor of the Parent class.

Example 2:

```
class ParentClass
{
        public ParentClass()
        {
                System.out.println("Parent Class default Constructor called");
        }
        public ParentClass(int val)
        {
                System.out.println("Parent Class parameterized Constructor, 
value: "+val);
        }
}
public class SubClass extends ParentClass
{
        public SubClass()
        {
                super();//Has to be the first statement in the constructor
                System.out.println("Child Class default Constructor called");
        }
        public SubClass(int val)
        {
                super(10);
                System.out.println("Child Class parameterized Constructor, 
value: "+val);
        }
        public static void main(String args[])
        {
                //Calling default constructor
                SubClass s = new SubClass();
                //Calling parameterized constructor
                SubClass s1 = new SubClass(10);
        }
}
```

Output

```
Parent Class default Constructor called
```

```
Child Class default Constructor called
Parent Class parameterized Constructor, value: 10
Child Class parameterized Constructor, value: 10
```

super.variable_name refers to the variable in the parent class

In the below given example we have the same variable in both parent and subclass.

```
class ParentClass
{
        int val=999;
}
public class SubClass extends ParentClass
{
        int val=123;

        public void disp()
        {
                System.out.println("Value is : "+val);
        }

        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.disp();
        }
}
```

Output

```
Value is : 123
```

Below given example will call only the **val** of the sub class. Without **super** keyword, you cannot call the **val** which is present in the Parent Class.

```
class ParentClass
{
        int val=999;
}
public class SubClass extends ParentClass
{
        int val=123;

        public void disp()
        {
                System.out.println("Value is : "+super.val);
        }

        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.disp();
        }
}
```

Output

```
Value is : 999
```

super.method_name refers to the method of the parent class

When you override the Parent Class method in the Child Class without super keywords support you will not be able to call the Parent Class method as shown in the below example:

```
class ParentClass
{
        public void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
{

        public void disp()
        {
                System.out.println("Child Class method");
        }

        public void show()
        {
                disp();
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.show();
        }
}
```

Output:

```
Child Class method
```

Here we have overridden the **Parent Class disp()** method in the SubClass and hence **SubClass disp()** method is called. If we want to call the **Parent Class disp()** method also means then we have to use the super keyword for it.

```
class ParentClass
{
        public void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
```

```
{

        public void disp()
        {
                System.out.println("Child Class method");
        }

        public void show()
        {
                //Calling SubClass disp() method
                disp();
                //Calling ParentClass disp() method
                super.disp();
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.show();
        }
}
```

Output

```
Child Class method
Parent Class method
```

When there is no method overriding then by default **Parent Class disp()** method will be called.

```
class ParentClass
{
        public void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
{
        public void show()
        {
                disp();
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.show();
        }
}
```

Output

```
Parent Class method
```

Polymorphism

Polymorphism is the ability to take more than one form. **Polymorphism** is one of the most important concept in OOPS (Object Oriented Programming Concepts). Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.
In **Java**, there are 2 ways by which you can achieve **polymorphic behavior**.

1. Method Overloading (Compile time Polymorphism (Static Binding))

2. Method Overriding (Run time Polymorphism (Dynamic Binding))

Runtime Polymorphism(Dynamic Binding)

Method Overriding in Java is the best example for **Runtime Polymorphism**. In this type of Polymorphism the **Parent class reference** can hold **object of Parent class or any sub class(Child class) of Parent.** This technique is called as **Dynamic Method Dispatch**

```
//Assigning Parent class Object to Parent class reference
Parent p = new Parent();
//Assigning Child class Object to Parent class reference
Parent p = new Child();
```

Dynamic Method Dispatch is a technique in which the overridden method to call is resolved at the **run-time** rather than at **compile time**. In this technique we will be assigning the **Child object** to the **Parent class reference** as depicted in the below code:

```
class Parent
{
    public void display(String name)
    {
        System.out.println("Welcome to Parent Class \""+name+"\"");
    }
    public void disp()
    {
        System.out.println("disp() method of Parent class");
    }
}
public class Child extends Parent
{
    public void display(String name)
    {
        System.out.println("Welcome to Child Class \""+name+"\"");
    }
    public void show()
    {
        System.out.println("show() method of Child class");
    }
    public static void main(String args[])
    {
        //Assign Child Object to Parent class reference
        Parent pp = new Child();
        pp.display("JIP");
     }
}
```

Here we have assigned Child class object (**new Child()**) to Parent class reference **(Parent pp)** and now come the question which of the **display()** method will be called. Here the **new Child()** object is resolved in the **run-time** the **display()** method of the **child** class will be called and hence the output will be

```
Welcome to Child Class "JIP"
```

What if the child class didn't override the parent class method?

In the above example of **Dynamic Method Dispatch** the **child** class has **overridden** the **parent** class **display()** method and hence the **child class display()** method is called, now the question is what will happen if the **child** class **didn't override** the **parent** class method.

```
class Parent
{
    public void display(String name)
    {
        System.out.println("Welcome to Parent Class \""+name+"\"");
    }
    public void disp()
    {
        System.out.println("disp() method of Parent class");
    }
}
public class Child extends Parent
{
    public void show()
    {
        System.out.println("show() method of Child class");
    }
    public static void main(String args[])
    {
        //Assign Child object to Parent class reference
        Parent pp = new Child();
        pp.display("JIP");
     }
}
```

When the **Child** class didn't override the **Parent** class method as in the above case then the **display()** method of the **Parent will be called**.

```
Welcome to Parent Class "JIP"
```

Can the Child class method be called in the Dynamic Method Dispatch?

No the child class method cannot be called in Dynamic method dispatch as shown below:

```
class Parent
{
    public void display(String name)
    {
        System.out.println("Welcome to Parent Class \""+name+"\"");
    }
    public void disp()
    {
        System.out.println("disp() method of Parent class");
    }
}
public class Child extends Parent
{
    public void display(String name)
    {
        System.out.println("Welcome to Child Class \""+name+"\"");
    }
    public void show()
    {
         System.out.println("show() method of child class");
    }
    public static void main(String args[])
    {
        //Assign Child refernce to Parent class
        Parent pp = new Child();
        pp.show();
     }
}
```

When we run the above code then we will get the below compile time exception
"The method show() is undefined for the type Parent".

In Dynamic Method Dispatch

- The **Overridden methods** of the **Child** class **can be called.**
- **Non-Overridden** methods of the **Parent** class **can be called.**
- **Child** class methods **cannot be called**.

Abstract Class in Java

The **abstract keyword** can only be used on **classes** and **methods** in **Java**. An abstract class **cannot be instantiated** and an abstract method can have **no implementation**. When a class is declared with **abstract keyword** then that particular class **cannot be instantiated**. It can only be extended and then all the methods of the abstract class needs to be implemented by the class which extends our abstract class

Example of abstract class

```
abstract class A{}
```

An abstract class can have both abstract and non-abstract methods as well.

Abstract Method in Java

When a method has abstract keyword then the class should definitely be an abstract class and when a method is declared abstract then it cannot have implementation.

Example of abstract method

```
abstract void disp();
```

Abstract classes and Abstract methods are like skeletons. It defines a structure, without any implementation.

```
abstract class A
{
    abstract void disp();
    public void show()
    {
        System.out.println("Show method"):
    }
}
```

Example of Abstract Method and Abstract **Class in Java**

In this example, we have abstract **class Car** and an abstract method **move()**, the implementation of the Car class is provided by **Bmw Class** and **Audi Class**.

```
abstract class Car
{
    public abstract void move();
}
class Bmw extends Car
{
    @Override
    public void move()
    {
        System.out.println("Move method of BMW");
    }

}
class Audi extends Car
{
    @Override
    public void move()
    {
        System.out.println("Move method of Audi");
    }

}
public class Logic
{
    public static void main(String args[])
    {
        Car c = new Bmw();
        c.move();
    }
}
```

Output

```
Move method of BMW
```

<u>Can Abstract Class have constructor?</u>

An abstract class can have constructor, abstract method, non abstract method , data member and even main method as well.

```
abstract class Shape
{
    Shape()
    {
        System.out.println("Shape constructor called");
    }
    abstract void color();
    void size()
    {
        System.out.println("Size method called");
    }

}
class Rectangle extends Shape
{
    void color()
    {
        System.out.println("Color of Rectange is Blue");
    }
}
public class Logic
{
    public static void main(String args[])
    {
        Rectangle rect = new Rectangle();
        rect.color();
        rect.size();
    }
}
```

Output

```
Shape constructor called
Color of Rectange is Blue
Size method called
```

Note:

- If we have declared a **method as abstract** then you must declare the **class as abstract**, abstract method cannot be present in a **non-abstract class.**
- Abstract Class can have **concrete methods(non-abstract)** as well.
- Abstract method should not have implementation(no body).
- Abstract method declaration should end with a **semicolon (;)**
- The class extending the abstract class should implement all the abstract methods.

INTERFACE:

An interface is a blueprint of a class, it can have methods and variables like a class but methods declared in an interface will be by **default abstract**(only declaration no body) and the variables declared will be **public, static & final by default.**

Use of Interface in Java

- Using interface we can achieve **100% abstraction** in **Java**, as the methods don't have body and the class needs to implement them before they access it.
- Java **does not support multiple inheritance. U**sing the interface we can achieve this as a class can implement more than one interface.

Example:

```
interface Shape
{
        public abstract void size();
}
public class Logic implements Shape
{
    @Override
    public void size()
    {
        System.out.println("Size method implementation called");
    }
    public static void main(String args[])
    {
        Logic l = new Logic();
        l.size();

        //Dynamic binding
        Shape s = new Logic();
        s.size();
    }
}
```

Output

```
Size method implementation called
Size method implementation called
```

In the above code we have an **interface "Shape"** which has a **abstract method "size()"** and Logic is the class which implements the size() method.

Can Interfaces have constructor?

Unlike Abstract class which can have constructor, non abstract method , main method. Interface **cannot have Constructor, non abstract method and main method.**

```
public interface Shape
{
       public Shape()
       {
           System.out.println("Constructor not allowed in Interface");
       }
        public abstract void size();
        public void disp()
        {
            System.out.println("non-abstract method not allowed in 
Interface");
        }
        public static void main(String args[])
        {
           //Some Logic
        }
}
```

The above interface will throw compilation errors as the interface cannot have constructor, non-abstract method and main method(as **public and abstract qualifiers** are only permitted).

Note:

- An Interface **cannot be instantiated** in Java.
- Methods declared in a interface should be **public and abstract**
- Interface cannot have **concrete methods(non-abstract methods or methods with body)**
- Variables declared should be **public, static & final** even though if you miss any one or all the qualifiers it will be automatically assigned. All valid scenarios of variable declaration are as shown below:

```
interface Test
{
int a = 10;
public int b = 10;
static int c = 10;
final int d = 10;
static final int e =10;
public static int f= 10;
public final int g =10;
}
```

- Interface variables must be initialized at the time of declaration else the compiler will throw error. The following declaration is invalid.

```
interface Test
{
int a;
}
```

- An Interface can only **extend** other interface (only one)
- A Class can **implement any number of interface**

```
interface Interface1
{
public void method1();
}
interface Interface2 extends Interface1
{
public void method2();
}
public class Demo implements Interface1,Interface2
{
 public void method2()
 {
 }
 public void method1()
 {
 }
 public static void main(String args[])
 {
 }
}
```

- If two interface have methods with **same signature and same return type** then the implementing class can implement any one of those.

```
interface Interface1
{
public void method1();
}
interface Interface2
{
public void method1();
}
public class Demo implements Interface1,Interface2
{
public void method1()
{
}
public static void main(String args[])
 {
 }
}
```

- If two interface have methods with **same signature and different return type** cannot be implemented at the same time

```
interface Interface1
{
  public void method1();
}
interface Interface2
{
 public int method1();
}
public class Demo implements Interface1,Interface2
{
 public void method1() //will throw compilation error
  {
  }
  public int method1() //will throw compilation error
{
}
public static void main(String args[])
 {
 }
}
```

- In the implementing class we **cannot change** the variable value which is declared in the interface as it is **final by default**

```
interface Interface1
{
int val=10;
public void method1();
}

public class Demo implements Interface1
{
public void method1()
{
}
public static void main(String args[])
{
 Interface1.val=20; //Will throw compilation error
}
}
```

SHADOWING:

A field is considered **shadowed** when

- a subclass of its declaring class declares a field with the **same name.**
- a variable having the same name and type is declared in the local scope.
- a method argument/parameter is declared with a same name and type.

Local variable shadowing:

```
public class MyClass
{
        private int count = 10;
        private void localVariable()
        {
                int count = 5;
                System.out.println("count = "+ count);
        }

        public static void main(String[] args)
        {
                MyClass test = new MyClass();
                test.localVariable();
        }
}
```

The above code will output count = 5 because the *count* **local variable** declared at **line 6** shadows the variable *count* declared at the class level. If we want to access the instance variable, we need to add **this** keyword as shown below.

```
private void localVariable()
{
        int count = 5;
        System.out.println("count = "+ this.count);

}
```

Method argument shadowing

```
private int count;
public void setCount(int count)
{
  this.count = count;
}
```

This keyword is mandatory to resolve the ambiguity. Without **this**, the compiler cannot know whether we are assigning the *count* method argument value to itself. If you remove **this** keyword, you would get a compilation warning anyway.

Superclass field shadowing

```
public class SuperClass
{
  protected String val = "SUPER_VAL";
  protected void display()
  {
            System.out.println("val = "+this.val);
  }
}

public class ChildClass extends SuperClass
{
  private String val;
  public ChildClass(String value)
 {
            this.val = value;
  }

  public static void main(String[] args)
  {
            ChildClass child = new ChildClass("CHILD_VAL");
            child.display();
  }
}
```

The execution gives:

```
val = SUPER_VAL
```

The *val* field has been declared in the **SuperClass** but is shadowed in the **ChildClass** because the latter declares another field with same **name and type**. Although the **ChildClass** has been instantiated with “CHILD_VAL”, the execution of *child.display()* gives you “SUPER_VAL”.

The reason is simple. When the child instance is created, there are 2 variables *val*. The one from **SuperClass** with value “SUPER_VAL” and the one from **ChildClass** with injected value “CHILD_VAL” through a constructor.

When the *display()* method is called, since it is defined in the **SuperClass**, it is the *val* field in the **context of SuperClass** which is used. Thus the output shows “SUPER_VAL”.

```
public class ChildClass extends SuperClass
{
        private String val;
        public ChildClass(String value)
       {
                this.val = value;
                super.val = value;
        }

        public static void main(String[] args)
        {
                ChildClass child = new ChildClass("CHILD_VAL");
                child.display();
        }
}
```

In the above-modified code, we force the value for the hidden *val* field in **SuperClass** with *super.val = value* and the output gives:

```
val = CHILD_VAL
```

Now we add another class in the hierarchy to see how super keyword can be combined with type casting.

```
public class AncestorClass
{
        protected String val = "ANCESTOR_VAL";
}

public class SuperClass extends AncestorClass
{
        protected String val = "SUPER_VAL";
}




public class ChildClass extends SuperClass
{
        private String val = "CHILD_VAL";
        public void displayVal()
        {
                System.out.println("val = " + super.val);
        }

        public static void main(String[] args)
        {
                ChildClass child = new ChildClass();
                child.displayVal();
        }
}
```

The output of the above program will be:
val = SUPER_VAL

Suppose we want to display the val value of ancestor class? Obviously, the super keyword only refers to the first parent class up in the class hierarchy. Casting can be used to display the value of the ancestor class.

```
public class ChildClass extends SuperClass
{
        private String val = "CHILD_VAL";
        public void displayVal()
        {
                System.out.println("val = " + ((AncestorClass) this).val);
        }

        public static void main(String[] args)
        {
                ChildClass child = new ChildClass();
                child.displayVal();
        }
}
```

Now the output will be val = ANCESTOR_VAL.

Static Blocks

Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. The code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class).Consider the following example:

```
class Test
{
 static int i;
 int j;
  // start of static block
  Static
  {
   i = 10;
   System.out.println("static block called ");
   }     // end of static block
  }
class Main
{
 public static void main(String args[])
 {
 // Although we don't have an object of Test, static block is  called because i is being accessed in
//following statement.
   System.out.println(Test.i);
  }
}
```

Output:

```
static block called

10
```

- Also, static blocks are executed before constructors. Consider the following example:

```
class Test
 {
  static int i;
   int j;
  static
  {
   i = 10;
     System.out.println("static block called ");

  }


  Test()
 {
 System.out.println("Constructor called");
  }
}
class Main
{
 public static void main(String args[])
  {
  // Although we have two objects, static block is executed only once.
  Test t1 = new Test();
  Test t2 = new Test();
 }
}
```

Output:

```
static block called
Constructor called
Constructor called
Static Block
```

- A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Example 1: Single Static Block

```
class JavaExample
{
  static int num;
  static String mystr;
  static
  {
    num = 97;
    mystr = "Static keyword in Java";
  }
  public static void main(String args[])
  {
    System.out.println("Value of num: "+num);
    System.out.println("Value of mystr: "+mystr);
  }
}
```

Output:

```
Value of num: 97
Value of mystr: Static keyword in Java
```

Example 2: Multiple Static Block

They execute in the given order which means the first static block executes before second static block. Thus, values initialized by first block are overwritten by second block.

```
class JavaExample2
{
  static int num;
  static String mystr;
  //First Static block
  Static
  {
    System.out.println("Static Block 1");
    num = 68;
    mystr = "Block1";
  }
 //Second static block
 Static
 {
    System.out.println("Static Block 2");
    num = 98;
    mystr = "Block2";
  }
 public static void main(String args[])
 {
    System.out.println("Value of num: "+num);
    System.out.println("Value of mystr: "+mystr);
  }
}
```

Output:

```
Static Block 1
Static Block 2
Value of num: 98
Value of mystr: Block2
```

Note:

- A static block is a block of code which contains code that executes at class loading time.
- A static keyword is prefixed before the start of the block.
- All static variables can be accessed freely
- Any non-static fields can only be accessed through object reference, thus only after object construction.
- multiple static blocks would execute in the order they are defined in the class.
- All static blocks executes only once per classloader.

Non Static Blocks

- The Non-static blocks are class level blocks which do not have any prototype.
- The need for a non-static block is to execute any logic whenever an object is created irrespective of the constructor.
- The Non-static blocks are automatically called by the JVM for every object creation in the java stack area.
- We can create any number of Non-static blocks in Java.
- There is no keyword prefix to make a block non-static block, unlike static blocks.

- In case of multiple non-static blocks, the block executes the order in which it is defined in the class.
- All static and non-static fields can be access freely.
- All non-static block executes every time an object of the containing class is created.

A typical non-static block looks like :

```
{
// non static block
}
```

Example:

```
public class NonStaticBlockTest {
  {
    System.out.println("First Non-Static Block"); // first non-static block
  }
  {
    System.out.println("Second Non-Static Block"); // second non-static block
  }
  {
    System.out.println("Third Non-Static Block"); // third non-static block
  }
  NonStaticBlockTest() {
    System.out.println("Execution of a Constructor"); // Constructor
  }
  public static void main(String args[]) {
    NonStaticBlockTest nsbt1 = new NonStaticBlockTest();
    NonStaticBlockTest nsbt2 = new NonStaticBlockTest();
  }
}
```

Output

```
First Non-Static Block
Second Non-Static Block
Third Non-Static Block
Execution of a Constructor
First Non-Static Block
Second Non-Static Block
Third Non-Static Block
Execution of a Constructor
```

EXCEPTION

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

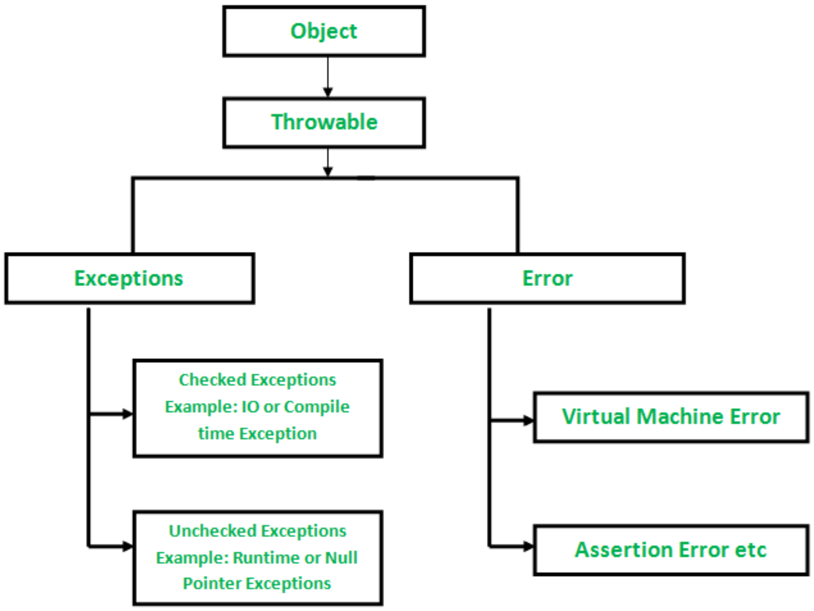
An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Exception Hierarchy:

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy.One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch,**Error** are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

- ***Checked Exceptions:*** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
  Example: Consider an example were we are reading a file myfile.txt and displaying its content on the screen. In this program there are three places where a checked exception is thrown. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.

```
import java.io.*;
class Example {
public static void main(String args[])
{
FileInputStream fis = null;
/*This constructor FileInputStream(File filename)
* throws FileNotFoundException which is a checked
* exception
*/
fis = new FileInputStream("B:/myfile.txt");
int k;

/* Method read() of FileInputStream class also throws
* a checked exception: IOException
*/
while(( k = fis.read() ) != -1)
{
System.out.print((char)k);
}

/*The method close() closes the file input stream
* It throws IOException*/
fis.close();
}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problems:
Unhandled exception type FileNotFoundException
Unhandled exception type IOException
Unhandled exception type IOException
```

There are two ways to resolve the checked compilation error:

- Using throws keyword
- Using try catch blocks

Method 1: Declare the exception using throws keyword:
Since all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. However, our code is throwing FileNotFoundException and IOException both but we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException so it by default covers that. We can also declare as shown:
public static void main(String args[]) throws IOException, FileNotFoundException.

```
import java.io.*;
class Example {
   public static void main(String args[]) throws IOException
   {
      FileInputStream fis = null;
      fis = new FileInputStream("B:/myfile.txt");
      int k;

      while(( k = fis.read() ) != -1)
      {
           System.out.print((char)k);
      }
      fis.close();
   }
}
```

Output:
File content is displayed on the screen.

Method 2: Handle them using try-catch blocks.
The approach we have used above is not the best exception handling practice. We should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

```
import java.io.*;
class Example {
   public static void main(String args[])
   {
        FileInputStream fis = null;
        try{
            fis = new FileInputStream("B:/myfile.txt");
        }catch(FileNotFoundException fnfe){
            System.out.println("The specified file is not " +
                         "present at the given path");
         }
        int k;
        try{
            while(( k = fis.read() ) != -1)
            {
                 System.out.print((char)k);
            }
            fis.close();
        }catch(IOException ioe){
            System.out.println("I/O error occurred: "+ioe);
         }
    }
}
```

Output:
File content is displayed on the screen.

Some other Checked Exceptions include –

- SQLException
- IOException
- ClassNotFoundException
- InvocationTargetException

- ***Unchecked Exceptions:*** Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of **RuntimeException** class.
  Example:

```
class Example {
   public static void main(String args[])
   {
    int arr[] ={1,2,3,4,5};
    /* My array has only 5 elements but we are trying to
         * display the value of 8th element. It should throw
     * ArrayIndexOutOfBoundsException
         */
    System.out.println(arr[7]);
   }
}
```

This code will compile successfully. However, at runtime it will show an ArrayIndexOutOfBounds Exception. Such a situation can be handled as shown:

```
class Example {
   public static void main(String args[]) {
    try{
       int arr[] ={1,2,3,4,5};
       System.out.println(arr[7]);
    }
      catch(ArrayIndexOutOfBoundsException e){
       System.out.println("The specified index does not exist " +
            "in array. Please correct the error.");
    }
   }
}
```

Output:
The specified index does not exist in array. Please correct the error.

Some other cases of unchecked exceptions include:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- IllegalArgumentException
- NumberFormatException

Types of Exceptions:

1. Built-in Exceptions

Built-in Exceptions	Description
ArithmeticException	It is thrown when an exceptional condition has occurred in an arithmetic operation.
ArrayIndexOutOfBoundsException	It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
ClassNotFoundException	This exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	An exception that is raised when a file is not accessible or does not open.
IOException	It is thrown when an input-output operation is failed or interrupted.
InterruptedException	It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
NoSuchFieldException	It is thrown when a class does not contain the field (or variable) specified.

2. User Defined Exceptions/ Custom Exceptions
   Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, a user can also create exceptions which are called 'User-Defined Exceptions'.

   - A user-defined exception must extend Exception class.
   - The exception is thrown using *throw* keyword.

   Example:

```
class MyException extends Throwable
{
 String str1;
 MyException(String str2) {str1=str2;}
 public String toString(){
 return ("MyException Occurred: "+str1);
}
}
class Example1
{
public static void main(String args[]){
 try{
      System.out.println("Start of try block");
      throw new MyException("Error Message");
     }
    catch(MyException exp)
    {
     System.out.println("Catch Block");
     System.out.println(exp);
    }
  }
}
```

Exception Handling / Catch or Specify Policy:

Catch or Specify Requirement or policy means that code that might throw certain exceptions must be enclosed by either of the following:

- A try statement that catches the exception and a handler for the exception, as described below.
- A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception as described below.

Java provides various methods to handle the Exceptions like:

- try
- catch

- finally
- throw
- throws

try block

The try block contains a set of statements where an exception can occur. It is always followed by a catch block, which handles the exception that occurs in the associated try block. A try block must be followed by catch blocks or finally block or both.

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

Nested try block

Example:

```
class Exception{
  public static void main(String args[]){
    try{
      try{
          System.out.println("going to divide");
          int b=59/0;
         }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(Throwable e)
         {System.out.println("Exception handeled");}
       System.out.println("casual flow");
    }
}
```

catch block

A catch block is where you handle the exceptions. This block must follow the try block and a single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in a try block, the corresponding catch block that handles that particular exception executes.

Multi-catch block:

Example:

```
public class SampleMultipleCatchBlock{
 public static void main(String args[]){
    try{
       int a[]=new int[5];
       a[5]=30/0;
      }
      catch(ArithmeticException e)
        {System.out.println("task1 is completed");}
      catch(ArrayIndexOutOfBoundsException e)
        {System.out.println("task 2 completed");}
      catch(Throwable e)
        {System.out.println("task 3 completed");}
      System.out.println("remaining code");
  }
}
```

finally block

A finally block contains all the crucial statements that must be executed whether an exception occurs or not. The statements present in this block will always execute, regardless an exception occurs in the try block or not such as closing a connection, stream etc.

```
class SampleFinallyBlock{
 public static void main(String args[]){
   try{
     int data=55/5;
     System.out.println(data);
    }
    catch(NullPointerException e)
       {System.out.println(e);}
    finally {System.out.println("finally block is executed");}
    System.out.println("remaining code");
  }
}
```

final vs finally vs finalize

final	Finally	finalize
It is a keyword.	It is a block.	It is a method.
Used to apply restrictions on class, methods & variables.	Used to place an important code.	Used to perform clean-up processing just before the object is garbage collected.
final class can't be inherited, method can't be overridden & the variable value can't be changed.	It will be executed whether the exception is handled or not.	–

throw vs throws

Throw	throws
1. Used to explicitly throw an exception	1. Used to declare an exception
2. Checked exceptions cannot be propagated using throw only	2. Checked exceptions can be propagated
3. Followed by an instance	3. Followed by a class
4. Used within a method	4. Used with a method signature
5. Cannot throw multiple exceptions	5. Can declare multiple exceptions

```
//Java throw example
void a()
{
  throw new ArithmeticException("Incorrect");
}

//Java throws example
void a()throws ArithmeticException
{
  //method code
}

//Java throw and throws example
void a()throws ArithmeticException
{
  throw new ArithmeticException("Incorrect");
}
```

CHAINED EXCEPTIONS:

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Constructors Of Throwable class Which support chained exceptions in java :

1. Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
2. Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

Methods Of Throwable class Which support chained exceptions in java :

1. getCause() method :- This method returns actual cause of an exception.
2. initCause(Throwable cause) method :- This method sets the cause for the calling exception.

Example of using Chained Exception:

```
// Java program to demonstrate working of chained exceptions
public class ExceptionHandling
{
    public static void main(String[] args)
    {
        try
        {
            // Creating an exception
            NumberFormatException ex =
                       new NumberFormatException("Exception");

            // Setting a cause of the exception
            ex.initCause(new NullPointerException(
                      "This is actual cause of the exception"));

            // Throwing an exception with cause.
            throw ex;
        }

        catch(NumberFormatException ex)
        {
            // displaying the exception
            System.out.println(ex);

            // Getting the actual cause of the exception
            System.out.println(ex.getCause());
        }
    }
}
```

Output:

```
java.lang.NumberFormatException: Exception

java.lang.NullPointerException: This is actual cause of the exception
```