Inheritance and Polymorphism

Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

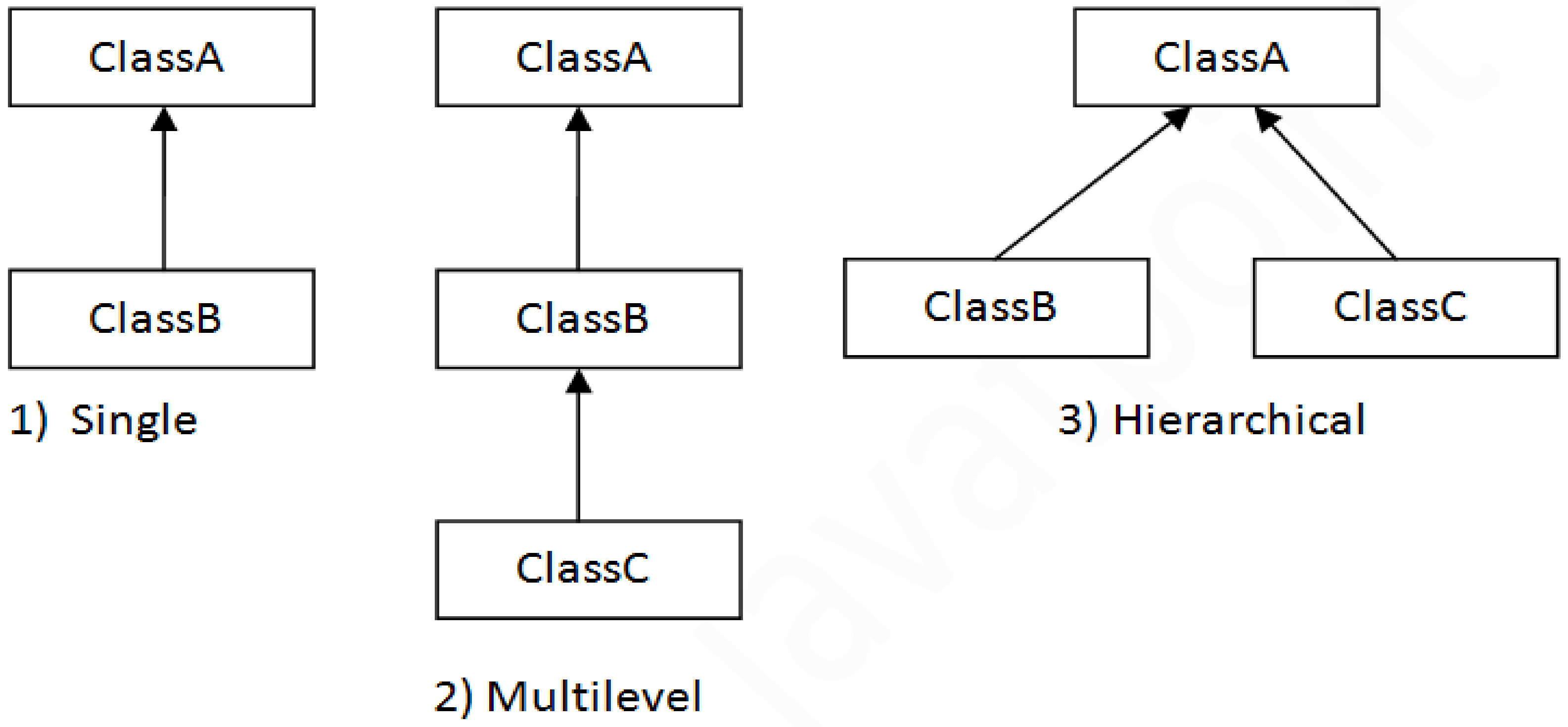
Basic Example of inheritance

```
class Employee
{
         float salary=40000;
}
class Programmer extends Employee
{
         int bonus=10000;
         public static void main(String args[]){
         Programmer p=new Programmer();
         System.out.println("Programmer salary is: "+p.salary);
         System.out.println("Bonus of Programmer is: "+p.bonus);
         }
}
```

In this example we are able to access the salary variable in the Programmer class.

Types of inheritance in java

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Single Inheritance Example

When a class inherits another class, it is known as a single inheritance.

Example of single level inheritance

```
class abc
{
   int x;
}
class Example extends abc
{
   int y;
   void assign_values(int x1,int y1)
   {
      x=x1;
      y=y1;
   }
   int sum()
   {
      return (x+y);
   }
```

```
public static void main(String[] args)
            {
Example ob = new Example();
ob.assign_values(12,34);
System.out.println("Sum is "+ob.sum());
            }
}
```

Output: Sum is 46

When there is a chain of inheritance, it is known as multilevel inheritance.

Example of Multilevel Inheritance

```
class abc
{
    int x;
}
class def extends abc
{
    int y;
}
class Example extends def
{
    int z;
    void assign_values(int x1,int y1,int z1)
    {
        x=x1;
        y=y1;
        z=z1;
    }
int sum()
    {
        return (x+y+z);
    }
        public static void main(String[] args)
        {
                Example ob = new Example ();
                ob.assign_values(12,34,56);
                System.out.println("Sum is "+ob.sum());
        }
      }
```

Hierarchical Inheritance

When two or more classes inherits a single class, it is known as hierarchical inheritance.

Example of Hierarchical Inheritance

```
//One base class
class A
{
   int x;
}
//Child class 1
class B extends A
{
   int y;
   void assign(int a,int b)
   {
      x=a;
      y=b;
   }
   int sum()
   {
      return x+y;
   }
}
```

```
//Child class 2
class C extends A
{
   int z;
   void assign(int a,int b)
   {
      x=a;
      z=b;
   }
   int product()
   {
      return x*z;
   }
}
public class Main
{
            public static void main(String[] args) {
            B obj1=new B();
            obj1.assign(2,3);
            System.out.println(obj1.sum());
            C obj2=new C();
            obj2.assign(4,5);
            System.out.println(obj2.product());
            }
}
```

Using the super Keyword

The keyword super refers to the superclass and can be used to invoke the superclass’s methods and constructors

- The keyword **super** refers to the superclass of the class in which **super** appears. It can be used in two ways:
- To call a superclass constructor.
- To call a superclass method..

Calling Superclass Constructors

A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited by a subclass. They can only be invoked from the constructors of the subclasses using the keyword super.

<u>The syntax to call a superclass's constructor is:</u>

super(), or super(parameters);

- The statement **super()** invokes the no-arg constructor of its superclass,
- The statement **super(arguments)** invokes the superclass constructor that matches the arguments.
- The statement **super()** or **super(arguments)** must be the first statement of the subclass's constructor;

This is the only way to explicitly invoke a superclass constructor.

Example 1 of super()

```
class Person
{
int id;
String name;
Person(int x,String y)
{
id=x;
name=y;
}
}
class Emp extends Person
{
float salary;
Emp(int a,String b,float c){
super(a,b);//reusing parent constructor
salary=c;
}
void display()
{
   System.out.println(id+" "+name+" "+salary);
}
}
```

```
class Main
{
public static void main(String[] args)
{
Emp e1=new Emp(1,"rohit",50000);
e1.display();
}
}
```

Calling super class method

```
class person
{
   int id;
   String name;
   void get_data(int x, String y)
   {
      id=x;
      name=y;
   }
}
class emp extends person
{
   float salary;
   void get_data(int a,String b,float c)
   {
      super.get_data(a,b);
      salary=c;
   }
   void show()
   {
      System.out.println(id+" "+name+" "+salary);
   }
}
```

```
class Main
{
      public static void main(String[] args)
      {
         emp e = new emp();
         e.get_data(1,"rohit",56000);
         e.show();

      }
}
```

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

Example

```
class A
{
void msg()
{
System.out.println("Hello");
}
}
class B
{
void msg()
{
System.out.println("Welcome");
}
}
class C extends A,B //suppose if it were
{
 public static void main(String args[])
{
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Polymorphism

The word polymorphism means having many forms. Polymorphism allows us to perform a single action in different ways.

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

<u>Compile time Polymorphism</u>

Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Example: Compile time Polymorphism(Method Overloading)

```
// Java program for Method overloading
class MultiplyFun {

static int Multiply(int a, int b)
{
                return a * b;
}
static double Multiply(double a, double b)
{
                return a * b;
}
}
 class Main {
                public static void main(String[] args)
                {
                System.out.println(MultiplyFun.Multiply(2, 4));
                System.out.println(MultiplyFun.Multiply(5.5, 6.3));
                }
           }
```

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to ***override*** the method in the superclass.
- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Example: Method Overriding

```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
```

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

Output:

```
k:3
```

Runtime Polymorphism in Java

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the **<u>object being referred to by the reference variable.</u>**

Explanation

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as **upcasting.** Java uses this fact to resolve calls to overridden methods at run time.

Example of method overridding/Dynamic Dispatch

```
class A
{
            void m1()
            {
System.out.println("Inside A's m1 method");
            }
}

class B extends A
{
            // overriding m1()
            void m1()
            {
System.out.println("Inside B's m1 method");
            }
}

class C extends A
{
            // overriding m1()
            void m1()
            {
System.out.println("Inside C's m1 method");
            }
}
```

```
// Driver class
class Dispatch
{
public static void main(String args[])
            {
            A a = new A();
            B b = new B();
            C c = new C();
            // obtain a reference of type A
            A ref;
            // ref refers to an A object
            ref = a;
            // calling A's version of m1()
            ref.m1();
            ref = b;
            ref.m1();
            ref = c;
            ref.m1();
            }
}
```

Explanation

The above program creates one superclass called A and it’s two subclasses B and C. These subclasses overrides m1( ) method.

- Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.A a = new A(); // object of type A B b = new B(); // object of type B C c = new C(); // object of type C
- Now a reference of type A, called ref, is also declared, initially it will point to null.A ref; // obtain a reference of type A
- Now we are assigning a reference to each type of object (either A’s or B’s or C’s) to ref, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined by the type of object being referred to at the time of the call.

```
ref = a; // r refers to an A object
ref.m1(); // calling A's version of m1()
```

In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

```
class A
{
		int x = 10;
}
class B extends A
{
		int x = 20;
}
public class Test
{
		public static void main(String args[])
		{
				A a = new B(); // object of type B
				System.out.println(a.x); //10 will be the output
		}
}
```

Advantages of runtime polymorphism

- Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.
- It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

Abstract class and Abstract method

- There are situations in which it is required to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method
- That is, sometimes we want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details
- This kind of class will be **abstract class** and method which will be just declared, but not defined in the superclass will be **abstract method.**
- We can declare reference of superclass type which is abstract but its object cannot be instantiated.
- Abstract class may contain non-abstract methods also.
- Abstract methods will be defined in the subclasses, which are inheriting the abstract classes

Example 1

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Output:

```
B’s implementation of callme
This is a concrete method
```

Example 2

```
// to show the working of abstract class
abstract class Shape
{
    int l,b;
    double r;
    abstract void area();
}
class Rectangle extends Shape
{
    Rectangle(int x,int y)
    {
        l=x;
        b=y;
    }
    int area()
    {
        return (l*b);
    }
}
class Circle extends Shape
{
    Circle(double y)
    {
        r=y;
    }
    void area()
    {
        System.out.println("Area of circle  is "+
(3.14*r*r));
    }
}
public class Test
{
public static void main(String[] args) {
Shape s;// Null
s= new Rectangle(12,34);
System.out.println(s.area());
s= new Circle(34.56);
s.area();
}
}
```

Upcasting and downcasting

- Upcasting: Upcasting is the typecasting of a **child object** to a **parent object**. Upcasting can be done implicitly. Upcasting gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature. Instead of all the members, we can access some specified members of the child class. For instance, we can access the overridden methods.

- Downcasting: Similarly, downcasting means the typecasting of a **parent object** to a **child object**. Downcasting cannot be implicitly**.**

Example

```
// Java program to demonstrate
// Upcasting Vs Downcasting
// Parent class
class Parent {
        String name;
        void method()
        {
        System.out.println("Method from Parent");
        }
}
// Child class
class Child extends Parent {
        int id;
        void method()
        {
        System.out.println("Method from Child");
        }
}
```

```
public class Main {
        public static void main(String[] args)
        {
                // Upcasting
                Parent p = new Child();
                p.name = “Hello";
                // This parameter is not accessible
                // p.id = 1;
                System.out.println(p.name);
                p.method();
                // Trying to Downcasting Implicitly
       // Child c = new Parent(); - > compile time
error
        // Downcasting Explicitly
                Child c = (Child)p;
                c.id = 1;
                System.out.println(c.name);
                System.out.println(c.id);
                c.method();
        }
}
```

Output:

```
Hello
Method from Child
Hello
1
Method from Child
```

More points on Downcasting

1st Scenario:

```
class ABC{}
class PQR extends ABC{
 public static void main(String args[]){
PQR obj=new ABC();//Compile time error
}
}
```

2nd Scenario:

```
class ABC{}
class PQR extends ABC{
 public static void main(String args[]){
ABC obj1=new ABC();
PQR obj2=(PQR) obj1;//Runtime error[ClassCastException]
}
}
```

3rd Scenario:

```
class ABC{}
class PQR extends ABC{
 public static void main(String args[]){
ABC obj1=new PQR();
PQR obj2=(PQR)obj1;
}
}
```

In third scenario, there will be no error

Instanceof operator

- The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

- The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Example 1:

```
class Simple1{
public static void main(String args[]){
Simple1 s=new Simple1();
System.out.println(s instanceof Simple1);//true
}
}
```

Example 2-An object of subclass type is also a type of parent class

```
class ABC{}
class PQR extends ABC{
 public static void main(String args[]){
 PQR obj=new PQR();
 System.out.println(obj instanceof ABC);//true
 }
}
```

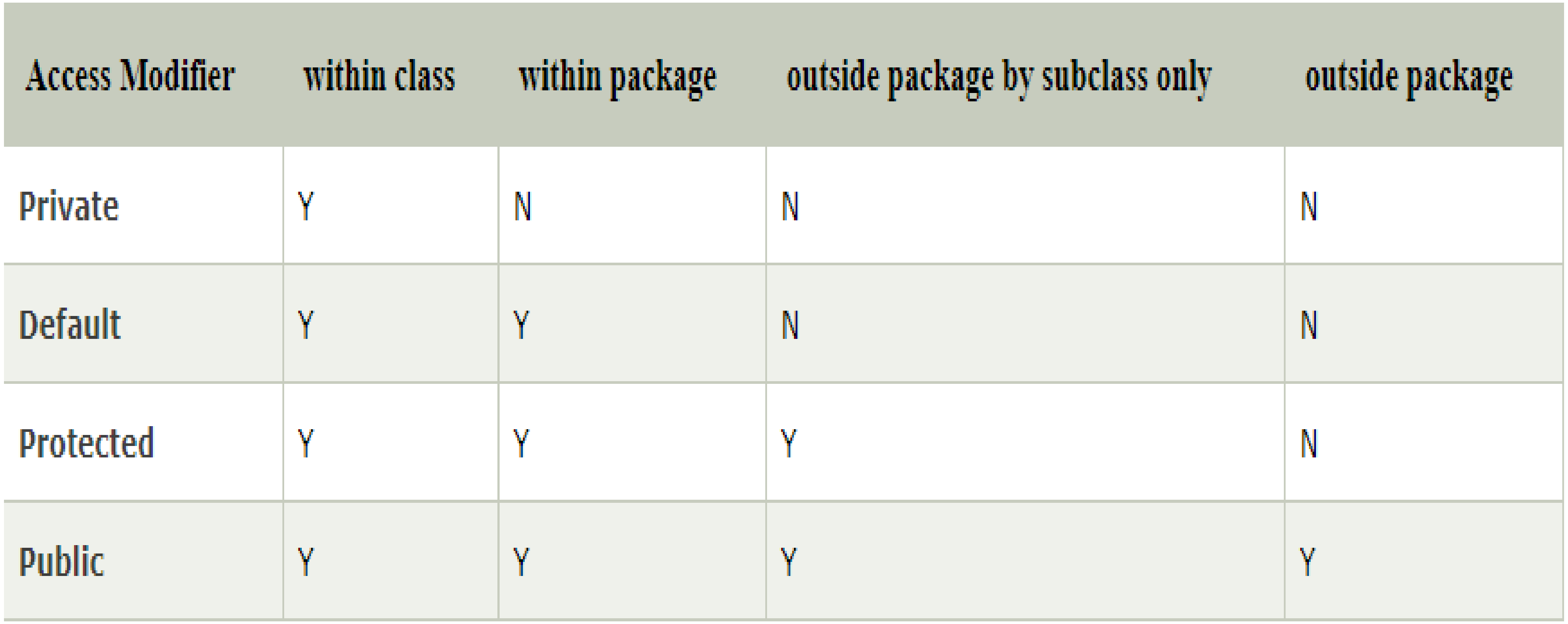
Example 3-instanceof can help in downcasting

```
class ABC{}
class PQR extends ABC{
 public static void main(String args[]){
ABC obj1=new PQR();
if(obj1 instanceof PQR) //It will return true
{
  PQR obj2=(PQR)obj1;
  System.out.println("Downcasting done");
}
else
{
System.out.println("Downcasting not possible");
 }
}
}
```

Access Modifiers/or levels in Java

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access modifiers-Summary



Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Q1

Which of these is correct way of inheriting class A by class B?

A. class B + class A {}

B. class B inherits class A {}

C. class B extends A {}

D. class B extends class A {}

Q2(Output??)

```
class A
{
    int i;
    void display()
    {
        System.out.println(i);
    }
}
class B extends A
{
    int j;
    void display()
    {
        System.out.println(j);
    }
}
```

```
class inheritance_demo
  {
      public static void main(String args[])
      {
          B obj = new B();
          obj.i=1;
          obj.j=2;
          obj.display();
      }
  }
```

A.0

B.1

C.2

D.Compile time error

Q3(Output??)

```
class A
  {
     int i;
  }
  class B extends A
  {
     int j;
     void display()
     {
        super.i = j + 1;
        System.out.println(j + " " + i);
     }
  }
```

```
class inheritance
  {
     public static void main(String args[])
     {
        B obj = new B();
        obj.i=1;
        obj.j=2;
        obj.display();
     }
  }
```

A.2 2

B.3 3

C.2 3

D.3 2

Q4(Output??)

```
class A
{
	A()
	{
	System.out.print("A ");
		}
}
class B extends A
{
	B()
	{
	   System.out.print("B ");
	}
}
class C extends B
{
	C()
	{
	   super();
	}
}
```

```
public class Main
{
		public static void main(String[] args)
{
		C obj=new C();
}
}
```

OP1: B   A

OP2: A   B

OP3: Compile time error

OP4: Blank output

Q5(Output??)

```
class ABC {
      int x;
}
class PQR extends ABC {
                int y;
}
public class Main {
            public static void main(String[] args)
            {
                        ABC obj = new PQR();
                        System.out.println(obj.y);
            }
}
```

A. 0
B. 1
C. Compile time error
D. Runtime error

Q6(Output??)

```
class ABC {
    int x;
}
class PQR extends ABC {
            void display(){
              System.out.println("Hi");
            }
}
public class Main {
        public static void main(String[] args)
        {
                ABC obj1=new PQR();
                PQR obj2 = (PQR)obj1;
                obj2.display();
        }
}
```

A. Hi
B. Compile time error
C. Runtime error
D. Blank Output

Q7(Output??)

```
class ABC {
	private int x=2;
}
class PQR extends ABC {
			int y=3;
}
public class Main {
		public static void main(String[] args)
		{
			PQR obj=new PQR();

		System.out.println(obj.x*obj.y);
		}
}
```

A. 6
B. Compile time error
C. Runtime error
D. 0

Q8(Output??)

```
class A{}
class B extends A{
 public static void main(String args[]){
A obj1=new B();
if(obj1 instanceof B)
{
B obj2=(B)obj1;
System.out.println("Hi");
}
else
{
System.out.println("Hello");
}
}
}
```

A. Hi
B. Hello
C. Compile time error
D. Runtime error

Programming in Java

this, super, static & final Keywords

Singleton class design pattern

Contents

‘final’ Keyword

- ‘final’ keyword is used to:
  - declare variables that can be assigned value only once (constant).

```
final type identifier = expression;
```

  - prevent overriding of a method.

```
final return_type methodName (arguments if any)
    {
        body;
    }
```

  - prevent inheritance from a class.

```
final class Class_Name
    {
     class body;
    }
```

final variable

```
public class Test
{
    public static void main(String[] args) {
      final int x=10;
      x=x+1;//Compilation error, as we are trying to modify the final variable(constant)
      System.out.println(x);
    }
}
```

final class(Non-inheritable class)

```
final class A
{


}
class B extends A // Compilation error [A is final and non-inheritable]
{


}
public class Test
{
      public static void main(String[] args) {
        B obj=new B();
      }
}
```

final method-It is used to prevent overriding

```
class A
{
   final void show()
   {
      System.out.println("A");
   }
}
class B extends A
{
   void show()//Compilation Error:show() in B cannot override show() in A(Overridden method is final)
   {
      System.out.println("B");
   }
}
public class Test
{
      public static void main(String[] args) {
        B obj=new B();
        obj.show();
      }
}
```

Immutable class

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

<u>Following are the requirements:</u>

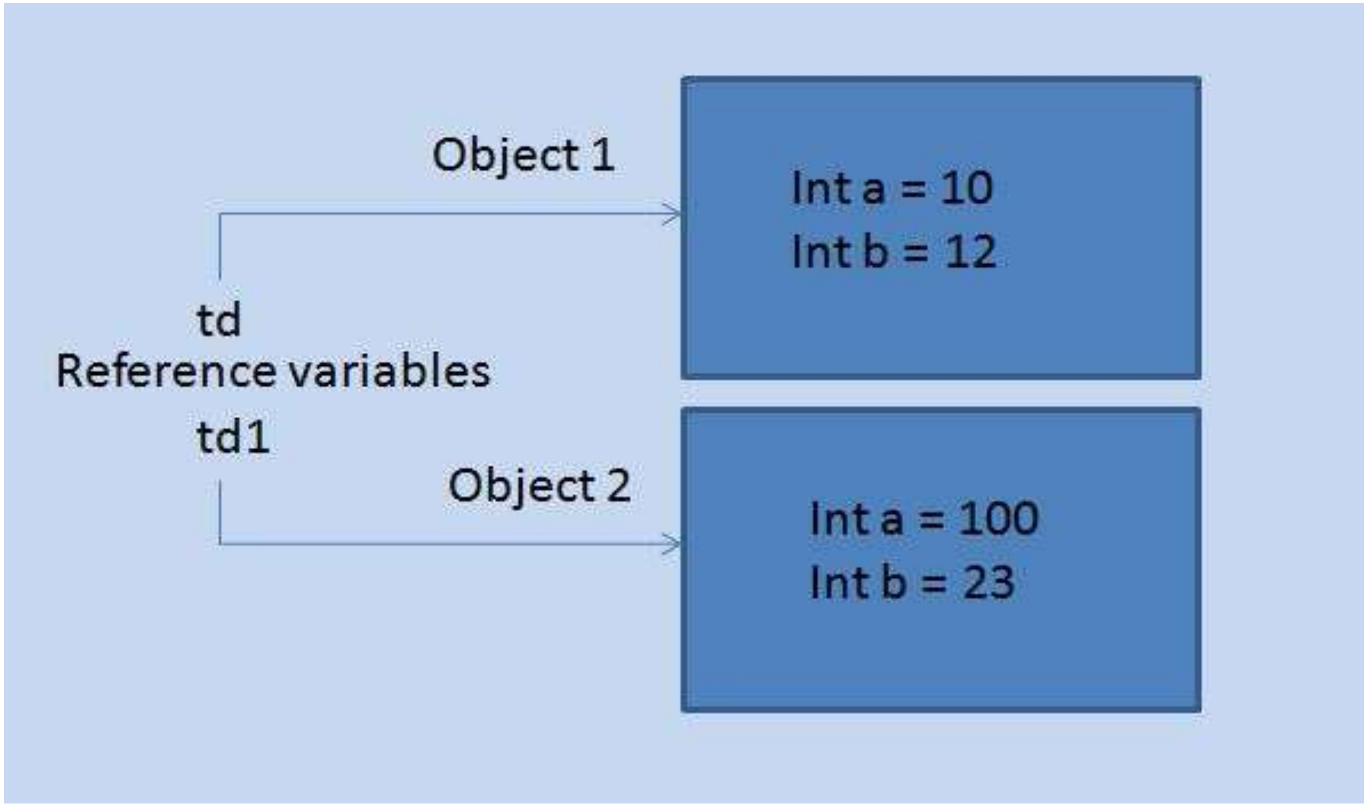
- The class must be declared as final (So that child classes can’t be created)
- Data members in the class must be declared as private (So that direct access is not allowed)
- Data members in the class must be declared as final (So that we can’t change the value of it after object creation)
- Only getters methods for accessing the values/ or parameterized constructors must be there to assign the value
- No setters should be there, so that there is no option to change the value of instance variables.

Example

```
final class Employee{
final String pancardNumber;
public Employee(String pancardNumber){
this.pancardNumber=pancardNumber;
}
public String getPancardNumber(){
return pancardNumber;
}
}
public class Main
{
    public static void main(String[] args)
    {
        Employee obj=new Employee("123");
        System.out.println(obj.getPancardNumber());
    }
}
```

‘this’ Keyword

- 'this' is used for pointing the current class instance.
- Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called.



```
class ThisDemo1{
                    int a = 0;
                    int b = 0;
                    ThisDemo1(int x, int y)
                       {
                          this.a = x;
                          this.b = y;
                        }

                    public static void main(String [] args)
                        {
                          ThisDemo1 td = new ThisDemo1(10,12);
                          ThisDemo1 td1 = new ThisDemo1(100,23);
                          System.out.println(td.a);
                          System.out.println(td.B);

System.out.println(td1.a);
                  System.out.println(td1.B);
                }
                  }
```

Chaining of constructors using this keyword

- Chaining of constructor means calling one constructor from other constructor.

- We can invoke the constructor of same class using ‘this’ keyword.
- Rules of constructor chaining :
  - ➢ The this() expression should always be the first line of the constructor.
  - ➢ There should be at-least be one constructor without the this() keyword.

Need of Constructor Chaining

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

Example

```
// Java program to illustrate Constructor Chaining
within same class Using this() keyword

public class Temp

{

        // default constructor 1

        // default constructor will call another
constructor

        // using this keyword from same class

        Temp()

        {

                // calls constructor 2

                this(5);

                System.out.println("The Default
constructor");

        }

// parameterized constructor 2

        Temp(int x)

        {

                // calls constructor 3

                this(5, 15);

                System.out.println(x);
```

```
        // parameterized constructor 3

        Temp(int x, int y)

        {

                System.out.println(x * y);

        }


        public static void main(String args[])

        {

                // invokes default constructor
first

                new Temp();

        }

}
```

Output:

```
75

5

The Default constructor
```

Constructor chaining using super

- **super()** keyword to call constructor from the base class.
- Constructor chaining occurs through **inheritance**. A sub class constructor’s task is to call super class’s constructor first. This ensures that creation of sub class’s object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.
- Note : Similar to constructor chaining in same class, **super()** should be the first line of the constructor as super class’s constructor are invoked before the sub class’s constructor.

Constructor Chaining using ‘super’

```
class A
{
    A(int x)
    {
        System.out.println(x);
    }
}
class B extends A
{
    B(int x,int y)
    {
        super(x);
        System.out.println(x+" "+y);
    }
}
class C extends B
{
    C(int x,int y,int z)
    {
        super(x,y);
        System.out.println(x+" "+y+" "+z);
    }
}
public class Main
{
         public static void main(String args[])
         {
                  C obj=new C(1,2,3);
         }
}
```

Output:

```
1
1 2
1 2 3
```

‘static’ Keyword

- used to represent class members
- Variables can be declared with the “static” keyword.

  static int y = 0;

- When a variable is declared with the keyword “static”, its called a “**class variable**”.
- All instances share the same copy of the variable.
- A class variable can be accessed directly with the class, without the need to create a instance.

- Note: There's no such thing as static classs. “static” in front of class creates compilation error.

static methods

- Methods can also be declared with the keyword “static”.
- When a method is declared static, it can be used without creating an object.

```
class T2 {
            static int triple (int n)
                  {return 3*n;}
            }

class T1 {
             public static void main(String[] arg)
                  {
                     System.out.println( T2.triple(4) );
                     T2 x1 = new T2();
                     System.out.println( x1.triple(5) );
                  }
             }
```

- Methods declared with “static” keyword are called “**class methods**”.
- Otherwise they are “**instance methods**”.
- **Static Methods Cannot Access Non-Static Variables.**
- The following gives a compilation error, unless x is also static.

```
class T2 {
        int x = 3;
        static int returnIt () { return x;}
      }


class T1 {
        public static void main(String[] arg) {
                System.out.println( T2.returnIt() ); }
     }
```

‘super’ Keyword

- ‘super’ keyword is used to:
  - invoke the super-class constructor from the constructor of a sub-class.

    super (arguments if any);

  - invoke the method of super-class on current object from sub-class.

    super.methodName (arguments if any);

  - refer the super-class data member in case of name-conflict between super and sub-class data members.

    super.memberName;

Important

‘this’ and ‘super’ both are non-static.

.

Singleton Class(or Singleton Design Pattern)

- In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.
- After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created.
- So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined. **<u>To design a singleton class:</u>**

▪ Make constructor as private.

▪ Write a static method that has return type object of this singleton class.

- **Singleton Design Pattern**
- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.

- The singleton class must provide a global access point to get the instance of the class.

Advantage:

Saves memory because object is not created at each request. Only single instance is reused again and again.

- Usage of Singleton Design Pattern

- Singleton pattern is mostly used in multi-threaded and database applications.

- It is used in logging, caching, thread pools, configuration settings etc.

Creating Singleton Design Pattern

To create the singleton class, we need to have static member of class, private constructor and static factory method.

- Static member: It gets memory only once because of static, it contains the instance of the Singleton class.
- Private constructor: It will prevent to instantiate the Singleton class from outside the class.
- public static factory method: This provides the global point of access to the Singleton object and returns the instance to the caller.

Example

```
// Classical Java implementation of singleton
      design pattern

class MySingleton

{

      private static MySingleton obj;

   public int x;

      // private constructor to force use of
      getInstance() to create Singleton object

      private MySingleton() {x=12;}

      public static MySingleton getInstance()

      {

        if (obj==null)

                 obj = new MySingleton();

        return obj;

      }

}
```

```
public class Main

{

   public static void main(String[] args)

   {

      MySingleton obj1=MySingleton.getInstance();

      MySingleton obj2=MySingleton.getInstance();

      System.out.println(obj1+" "+obj2);//Same reference for obj1 and obj2

      System.out.println(obj1.x+" "+obj2.x);//12 12

      obj1.x=23;

      obj2.x=45;

      System.out.println(obj1.x+" "+obj2.x);//45 45

   }

}
```

Explanation

Here we have declared getInstance() static so that we can call it without instantiating the class. The first time getInstance() is called it creates a new singleton object and after that it just returns the same object. Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation.