Object Oriented Programming [Java]

**Components Of OOPS**

Encapsulation –

Binding (or wrapping) code and data together into a single unit i.e a class is known as encapsulation. It lets us hide the implementation details using different access modifiers.

**Objects**

Objects are the real world entities about which we code. Objects have properties and they perform functions. For example in a student management system the real world entities about which the system revolves are – students, instructor, course, batch etc.

**A Class**

Class is a template or a blue print and the objects are specific copies of it. For example a Vehicle class might look like :

public class Vehicle {

public String brand;

protected String model;

private double price;

int numWheels;

int yearOfManufactor;

String color;

public double getPrice(){

return price;

}

public void printDescription(){

System.out.println(brand +" " + model +" "+price+" "+numWheels);

}

}

Now each vehicle will be a specific copy of this template. The syntax to create an object of vehicle is as follows :

public static void main(String args[]){

Vehicle v = new Vehicle();

//v.fieldName – will give access to this vehicle’s fields

}

**Constructors**

Constructor is a special method that is called when an object is instantiated i.e created. It is used to initialize an object. Its name is same as the class name. Even though in the above vehicle example we haven’t created an explicit constructor there is a default constructor implicitly there. We can create our own constructor as well. Also we can define multiple constructors in a class as well.

public Vehicle(Double price){

this.price = price;

}

Here this is a keyword that refers to current object, So this.price refers to the data member (i.e. price) of this object and not the argument variable price.

One important point to note here is that as soon as we create our constructor the default constructor goes off.

Now when we have defined the above constructor and if it is the only constructor in the class, then we can’t create any object of Vehicle without giving its price. In a way we can actually restrict users that they can’t create a vehicle without giving its price. We can have more than one constructors within the same class (i.e constructor overloading), which constructor will be called will be decided on runtime depending on the type and number of arguments specified while creating the object.

Inheritance –

Inheritance is a mechanism by which one class can extend functionality from an existing class. It provides code reusability. The derived class inherits the states and behaviors from the base class. The derived class can add its own additional variables and methods. Syntax for inheritance is shown below –

public class Car extends Vehicle {

private int numDoors;

String company;

public int numDoors(){

return numDoors;

}

}

Here car (sub class) extends Vehicle (base class / super class) since every car is a vehicle. So car will now have all the properties and functions that a vehicle has except the private fields of vehicle class(since they are not inherited , but they can be accessed via functions of base class that aren’t private). What if both the base class and sub class have function with same signature i.e same name and arguments ? Say even car has a printDescription function as in vehicle.

public void printDescription(){

System.out.println("Car :" + company +" " + model +"

"+getPrice()+" "+numDoors);

}

then

Car c = new Car();

c.printDescription(); // This will call car’s printDescription

If we wish to call base class printDescription inside Car’s printDescription then “super “ keyword should be used.

super.printDescription(); // This will call Vehicle’s printDescription

Suppose Vehicle has one constructor as shown below :

public Vehicle(Double price){

this.price = price;

}

then Car, which extends Vehicle needs to have a constructor that passes value to the vehicle constructor which is implicitly called when we create an object of car.

public Car(double price){

super(price); // should be the first line

numWheels = 4;

company = "";

}

**Access Modifiers**

**Private :**

The private access modifier is accessible only within class.

class A{

private int data=40;

private void msg(){

System.out.println("Hello java");

}

}

public class Simple{

public static void main(String args[]){

A obj=new A();

System.out.println(obj.data);//Compile Time Error

obj.msg();//Compile Time Error

}

}

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

**Role of Private Constructor**

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

class A{

private A(){}//private constructor

void msg(){

System.out.println("Hello java");

}

}

public class Simple{

public static void main(String args[]){

A obj=new A();//Compile Time Error

}

}

**Default :**

When we explicitly don’t write any modifier it is default . This modifier is package friendly i.e it can be accessed within the same package.

package pack;

class A{

void msg(){System.out.println("Hello");}

}

package mypack;

import pack.\*;

class B{

public static void main(String args[]){

A obj = new A();//Compile Time Error

obj.msg();//Compile Time Error

}

}

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package. In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

**Protected :**

It is accessible within the same package and outside the package but only through inheritance.

package pack;

public class A{

protected void msg(){

System.out.println("Hello");

}

}

package mypack;

import pack.\*;

class B extends A{

public static void main(String args[]){

B obj = new B();

obj.msg();

}

}

Output:Hello

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

**Public :**

It is accessible everywhere.

package pack;

public class A{

public void msg(){

System.out.println("Hello");

}

}

package mypack;

import pack.\*;

class B{

public static void main(String args[]){

A obj = new A();

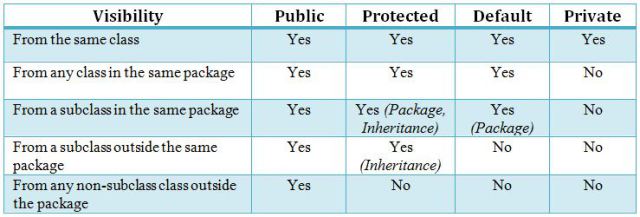
obj.msg();

}

}

Output:Hello

An important point to note here is that its better to make a variable private and then provide getters and setters in case we wish allow others to view and change it than making the variable public. Because by provding setter we can actually add constraints to the function and update value only if they are satisfied.



Polymorphism –

1. Polymorphism means to process objects differently based on their data type.
2. In other words it means, one method with multiple implementation, for a certain class of action. And which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the object)
3. This can be implemented by designing a generic interface, which provides generic methods for a certain class of action and there can be multiple classes, which provides the implementation of these generic methods. Lets us look at same example of a car. A car have a gear transmission system. It has four front gears and one backward gear. When the engine is accelerated then depending upon which gear is engaged different amount power and movement is delivered to the car.

Polymorphism could be static and dynamic both. Overloading is static polymorphism while, overriding is dynamic polymorphism.

Overloading in simple words means two methods having same method name but takes different input parameters. This called static because, which method to be invoked will be decided at the time of compilation

Overriding means a derived class is implementing a method of its super class.

**Method Overriding**

Ability of a variable to take different forms – A base class’ reference can refer to the object of its sub class i.e we can do something like this –

Vehicle v = new Car(1000);

Since every car is a vehicle so a vehicle(i.e. reference of type vehicle) can refer to a car. And not just the car, reference “v” here can refer to object of any other class that extends vehicle. But through this refernce “v” we can access only those properties of car which even a vehicle has i.e.

v.numDoors = 4; // This will give error as numDoors is

// car’s specific property

When both base class and sub class have functions of same signature then base class’ function is overriden by the subclass’ function.

Vehicle v1 = new Vehicle();

Vehicle v2 = new Car(1000);

v1.printDescription(); // Will call vehicle's printDescription

v2.printDescription(); // Will call car's printDescription

In case of v2, which printDescription should be called is decided on runtime (Runtime Polymorphism) based on the type of the object and not the type of reference. Same is the case with abstract class, a reference of abstract class can refer to objects of all its sub classes which themselves aren’t abstract.

Ability of a function to work with parameters of subtypes –

public void print(Vehicle v){

v.printDescription();

}

This print function expects a vehicle, so we can pass a car(or object of any of its subtype) to it i.e.

Car c = new Car();

print(c);

**Method Overloading**

Ability of a function to behave differently on basis of different parameters.

public int add(int a,int b){

return a+b;

}

public double add(double a,double b){

return a+b;

}

public char add(char a,char b){

return (char)(a+b);

}

Amongst these three add functions which add will be called finally, will be decided on runtime based on the type of parameters.

**Constructor Overloading**

Constructor overloading is similar to function overloading. At runtime while creating an object the number and type of parameters passed will decide that which constructor will be called.

public Vehicle(String color, double price){

this.color = "white";

this.price = price;

}

public Vehicle(double price){

this.price = price;

}

public Vehicle(){

}

**Modifiers**

**Static and Non-Static :**

Static properties are those that belong to the class rather each specific object. So their separate copies aren’t created. They are shared by all the objects of the class. You need to write static keyword before it in order to make it static.

class Student{

int rollno;

String name;

static String college="ITS";

}

Whereas properties like name, rollno can have different values for each object and are object specific and thus are non static. Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All student have its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

**Static methods**

Define static methods in the following scenarios only:

1. If you are writing utility classes and they are not supposed to be changed.
2. If the method is not using any instance variable.
3. If any operation is not dependent on instance creation.
4. If there is some code that can easily be shared by all the instance methods, extract that code into a static method.
5. If you are sure that the definition of the method will never be changed or overridden. As static methods can not be overridden.

class Student{

int rollno;

String name;

static String college = "ITS";

static void change(){

college = "BBDIT";

}

Student(int r, String n){

rollno = r;

name = n;

}

void display (){System.out.println(rollno+" "+name+" "+college);}

public static void main(String args[]){

Student.change();

Student s1 = new Student9 (111,"Karan");

Student s2 = new Student9 (222,"Aryan");

Student s3 = new Student9 (333,"Sonoo");

s1.display();

s2.display();

s3.display();

}

}

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

Note

1. Static methods can only access static variables and other static methods.
2. Non static members can only be accessed by making instance of that class.
3. Non static methods can access static as well as non static members.

Why java main method is static?

Because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

**Final Keyword :**

Final keyword can be applied before a variable, method and a class. A final variable is one whose value can’t be changed. So we can either initialise a final variable at the time of declaration or in a constructor. A final method is one that can’t be overriden. Making a class final means it can’t be inherited. (E.g : String class in java is final)

Final method is inherited but you cannot override it. For Example:

class Bike{

final void run(){System.out.println("running...");}

}

class Honda2 extends Bike{

public static void main(String args[]){

new Honda2().run();

}

}

Output:running...

**Final arrays in Java**

class Test

{

public static void main(String args[])

{

final int arr[] = {1, 2, 3, 4, 5}; // Note: arr is final

for (int i = 0; i < arr.length; i++)

{

arr[i] = arr[i]\*10;

System.out.println(arr[i]);

}

}

}

Output:

10

20

30

40

50

The array arr is declared as final, but the elements of array are changed without any problem. Arrays are objects and object variables are always references in Java. So, when we declare an object variable as final, it means that the variable cannot be changed to refer to anything else. For example, the following program 1 compiles without any error and program 2 fails in compilation.

// Program 1

class Test

{

int p = 20;

public static void main(String args[])

{

final Test t = new Test();

t.p = 30;

System.out.println(t.p);

}

}

Output: 30

// Program 2

class Test

{

int p = 20;

public static void main(String args[])

{

final Test t1 = new Test();

Test t2 = new Test();

t1 = t2;

System.out.println(t1.p);

}

}

Output: Compiler Error: cannot assign a value to final variable t1

So a final array means that the array variable which is actually a reference to an object, cannot be changed to refer to anything else, but the members of array can be modified.

**Abstract :**

An abstract method is one which does not have implementation. E.g

abstract void printStatus();//no body

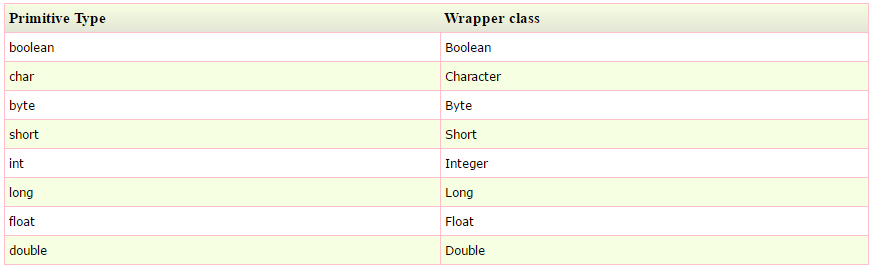
A class having even one abstract method has to be declared abstract, and since an abstract class is incomplete so you cannot create an instance of abstract class, but it can be extended. Also we can create reference of an abstract class. We will discuss more about in polymorphism.

Wrapper class in Java

Wrapper class in java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

The eight classes of java.lang package are known as wrapper classes in java. The list of eight wrapper classes are given below:



**Number Methods**

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.

Following is the list of some of the instance methods that all the subclasses of the Number class implements −

1. **xxxValue()** - Converts the value of this Number object to the xxx data type and returns it.
2. **compareTo()** - Compares this Number object to the argument.
3. **equals()** - Determines whether this number object is equal to the argumen
4. **valueOf()** - Returns an Integer object holding the value of the specified primitive.
5. **toString()** - Returns a String object representing the value of a specified int or Integer.
6. **parseInt()** - This method is used to get the primitive data type of a certain String.
7. **abs()** - Returns the absolute value of the argument.

Example: Using compareTo() with Integer and String

