CL309
Object Oriented Analysis and Design Lab
Lab # 03

Object-Oriented Programming: Classes and Members

Objectives
- Introducing classes and objects and differences between them.
- Understanding class members and their properties.
- Understanding static functions and variables.
- Knowing about method and constructor overloading.
- Passing objects in parameters.
- Understanding pass-by-value and pass-by-reference.
- Understanding THIS pointer

3.1 Writing Classes in Java
So far, you have been writing a lot of classes, but these were far from the idea of representing real-life objects, their rule was only to include the main method that we use to run the program. We will start now to write classes that represent real-life objects, these classes does not necessarily have main method and hence are not executable.

```
class Circle{
        //Radius is enough to determine all circle attributes
        public double radius;

        //Constructor
        public Circle(double r){
                radius = r;
        }
        //Methods
        public double getRadius(){
                return radius;
        }
        public double getArea(){
                return (radius * radius) * Math.PI;
        }
        public double getCircumference(){
                return 2 * Math.PI * radius;
        }
        public double getDiameter(){
                return radius * 2;
        }
}
```

The above class contained members declared using public keyword, this keyword means that the member is accessible from outside the class, that is, when we declare an object from the class Circle, we can access all public members, even if we declare the object outside Circle class. You can also notice a method that holds the same name of the class and does not has a return value, actually, we call it *constructor*. Constructors are used to determine the initial values necessary to build objects and are used after new keyword in declaration. For example, when declaring a Circle object, we need to specify the radius of that circle, because all circle measurements are based on the radius, so we write a constructor that takes radius as double. If we don't write a constructor for our class, Java creates a *default constructor* that takes no parameters, but once we write a constructor, the default constructor is no more available. To declare a Circle object, we use something similar to variable declaration.

```
Circle c1 = new Circle(10);
Circle c2 = new Circle(12);
System.out.println("c1 Area is " + c1.getArea());
System.out.println("c2 Diameter is " + c2.getDiameter());
```

As shown above, declaring objects is straight forward, all you have to do is to specify class name from which you create the object followed by object name. Then you initialize the object using new keyword followed by a constructor with necessary parameters. When we call a member from the object, we use dot '.' to access the members then specify member name. As we declared radius to be public, we can access it just like any other members to change the radius after declaration.

```
Circle c1 = new Circle(10);
System.out.println("c1 Area is " + c1.getArea());
c1.radius = 15;
System.out.println("c1 Area is now " + c1.getArea());
```

3.2 Different Properties of Class Members

By properties of members, we mean set of access rules that controls how these members are accessed and called from the class or an object of that class, we have introduced one of these properties in Circle class which is public. Here is a list of access rules and their use:

- **public**: states that the member is accessible from outside the class (from another class).
- **private**: in the contrary of public, private states that the member is accessible only from within the same class (a method or constructor inside the class).
- **static**: by static, we specify that a certain member can be called without declaring an object (by calling class name followed by dot then member name). This is clear when declaring main method, because when we run our class we do not declare an object from it.

It is important to know that static members cannot by called from non-static members and vice versa. Thats why we were declaring methods using static keyword, because main is static, we have to declare any method we wish to call from main as static also. On the variable level, the static variable is shared among all objects of the class. For now, these properties are enough for us.

The following example clearly explains the difference between public and private members:

```
class A{
        private int value = 10;
        public int getValue(){ return value; }
        public void setValue(int newValue){ value = newValue; }
}
class B{
        public static void main(String[] args){
        A a = new A();
        System.out.println(a.value); //Error, value is private
        }
}
```

Trying to access private member from outside the class reports compile-time error. It is a common practice in Object-Oriented that we declare members that store values as private and create *set* and *get* methods to access them, set methods are called *mutators*, and get methods are called *accessors*. In the above example, we can use accessors and mutators methods o access the value stored in class A as the following:

```
class B{
        public static void main(String[] args){
                A a = new A();
                a.setValue(33); //Changes the value by mutator
                method
                System.out.println(a.getValue()); //prints 33
        }
}
```

The following example shows the difference between static and non-static variables:

```
class StaticTest{
        private static int a;
        private int b;
        public StaticTest(){
                a = 0; b = 0;
        }

        //Accessors
        public static int getA(){ return a; }
        public int getB(){ return b; }

        //Mutators
        public static void setA(int newA){ a = newA; }
        public void setB(int newB){ b = newB; }
}
class Main{
        public static void main(String[] args){
        StaticTest s1 = new StaticTest();
        StaticTest s2 = new StaticTest();
        System.out.println("s1 values:");
        System.out.println("a =" + s1.getA() + ", b =" + s1.getB());
        System.out.println("s2 values:");
        System.out.println("a =" + s2.getA() + ", b =" + s2.getB());
        s1.setA(5);
        s1.setB(3);
        System.out.println("\ns1 values:");
        System.out.println("a =" + s1.getA() + ", b =" + s1.getB());
        System.out.println("s2 values:");
        System.out.println("a =" + s2.getA() + ", b =" + s2.getB());
        }
}
```

Compiling and running the above code shows the following output:

```
s1 values:
a =0, b =0
```

```
s2 values:
a =0, b =0

s1 values:
a =5, b =3
s2 values:
a =5, b =0
```

So what can be concluded from the output? The value of b is different for each object, changing the value of b in object s1 did not affect the value of b in s2, which is not the situation with a, because changing the value of a in s1 had its effect in a when called from s2, that is because static members are shared among all objects as stated, variable a and methods related to it could be called from the class name directly as shown below:

```
StaticTest.setA(13);

System.out.println(s1.getA() + " " + s2.getA());
```

In the above code, println method will print 13 two times. Remember to declare static methods when dealing with static variables, back to StaticTest class definition, for example, you can notice that getA and setA methods are declared static.

3.3 Methods and Constructors Overloading

Overloading means to have more than one constructor or more than one method having the same name, because there are more than one way to discriminate between methods, including name and number and type of parameters. To see how parameters can help in overloading, consider the following constructors.

```
class Rectangle{
        private double height, width;
        public Rectangle(double h, double w){
                height = h;
                width = w;
        }
        public Rectangle(double d){
                height = width = d;
        }
        public Rectangle(){
                height = width = 1;
        }
}
```

As the above example illustrates, we have three constructors having the same name, when we call a constructor after new keyword, depending on parameters we specify a matching constructor is selected.

```
Rectangle a = new Rectangle();//Third constructor
Rectangle b = new Rectangle(5);//Second constructor
Rectangle c = new Rectangle(3, 4);//First constructor
```

It is important to realize that discriminating between constructors and methods holding the same name are:

1. Number of parameters.
2. Order of parameters.
3. Types of parameters.

So it is important to know that parameter name has nothing to do with overloading, the following example shows this.

```
//This code generates error message.
class A{
        private int value;
                public void setValue(int t){
                value = t;
        }

        public void setValue(int s){
                value = s;
        }
}
```

To better understand classes and objects, lets analyze a well-known statement we've been using repeatedly:

```
System.out.println("Hello World");
```

System is a name of a class, inside System class, there is a public and static member (object) of class PrintWriter, the name of this object is out. Inside PrintWriter class, there is a public, non-static method called println that has several overloads.

3.4 Using Objects as Parameters

So far we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        // return true if o is equal to the invoking object
        boolean equals(Test o) {
                        if(o.a == a && o.b == b) return true;
                        else return false;
                }
        }

class PassOb {
        public static void main(String args[]) {
                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);
                System.out.println("ob1 == ob2: " + ob1.equals(ob2));
                System.out.println("ob1 == ob3: " + ob1.equals(ob3));
        }
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

As you can see, the equals( ) method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns true. Otherwise, it returns false. Notice that the parameter o in equals( ) specifies Test as its type. Although Test is a class type created by the program, it is used in just the same way as Java's built-in types.

3.5 A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed.

In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Simple types are passed by value.
class Test {
        void meth(int i, int j) {
                i *= 2;
                j /= 2;
        }
}
class CallByValue {
        public static void main(String args[]) {
                Test ob = new Test();
                int a = 15, b = 20;
                System.out.println("a and b before call: " +
                a + " " + b);
                ob.meth(a, b);
                System.out.println("a and b after call: " +
                a + " " + b);
        }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside meth( ) have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program:

```
// Objects are passed by reference.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        // pass an object
        void meth(Test o) {
             o.a *= 2;
             o.b /= 2;
             }
        }

class CallByRef {
        public static void main(String args[]) {
                Test ob = new Test(15, 20);
                System.out.println("ob.a and ob.b before call: " +
                ob.a + " " + ob.b);
                ob.meth(ob);
                System.out.println("ob.a and ob.b after call: " +
                ob.a + " " + ob.b);
        }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside meth( ) have affected the object used as an argument.

3.6 Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the incrByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
        int a;
        Test(int i) {
                a = i;
        }
        Test incrByTen() {
                Test temp = new Test(a+10);
                return temp;
        }
}

class RetOb {
public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
        + ob2.a);
        }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

As you can see, each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: since all objects are dynamically allocated using new, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

3.7 This pointer

When we wish to specify a member from the class, we can use this keyword, this refers to the current object of the class. For example, to discriminate between local and global variables in a class that have the same name, we use this to specify which one is in the scope of whole class.

```
class Square{
        private int sideLength;
        public Square(int sideLength){
                this.sideLength = sideLength;
        }
        public void setSideLength(int sideLength){
        //global local
                this.sideLength = sideLength;
        }
}
```

As you can see, we have two variables called sideLength, one is global for the whole class and it is where we store the value, and other is local for the constructor (or method). When we call this., we are returning back to the scope of the class, so if we are to specify any member after this, it has to be a global member.

Another use of this keyword is to call a constructor from within another constructor, this itself represents the call, so what you have to do is to pass parameters to this just like the way you pass them to any constructor. This technique is common when writing default constructor that does not require user to specify parameters.

```
class Vector{
        private double length;
        private double angle;
        public Vector(double length, double angle){
                this.length = length;
                this.angle = angle;
        }
        public Vector(){
        /* Calling the previous constructor,
        passing 1 as length and 0 as angle */
                this(1, 0);
        }
}
```

If you need to specify additional statements in the constructor from which you are calling another constructor, these statements should be specified after this call, in other words, when using this to call another constructor, it should be in the first statement.

Tasks

1. Class Tasks

Task1

Write a class that represents triangle named Triangle, the class must have the following members:

- ○ private double height;//Height
- ○ private double base;//Base length
- ○ public Triangle(double h, double b);//Constructor
- ○ public void setHeight(double x);//Sets height
- ○ public double getHeight();//Gets height
- ○ public void setBase(double x);//Sets base length
- ○ public double getBase();//Gets base length
- ○ public double getArea();//Returns the area of the triangle

Create a Test Application which determines Area of a triangle with height=3, base=4.

Task 2

Create a class called Date that includes three pieces of information as instance variables - a month (type int), a day (type int) and a year (type int). Your class should have a constructor that initializes the three instance variables and assumes that the values provided are correct.

1. Provide a set and a get method for each instance variable.
2. Provide a method “displayDate()” that displays the month, day and year separated by forward slashes (/).
3. Write a method “equals()” that takes a Date object, compares it with this Date, and returns a Boolean value.
4. Write a method “differenceDates()” that takes a Date object as an argument, subtracts the corresponding elements from this Date, and returns a Date object which is then displayed in the test program.

Write a test application named DateTest that demonstrates class Date’s capabilities.

2. Home Tasks

Task 3

Write a class Employee that represents an employee of some organization, the class should contain the following members:

- ○ private int id;//Employee id
- ○ private String name;//Employee name
- ○ private int type;//1 = employee, 2 = manager
- ○ private double Salary;//Employee salary
- ○ public Employee(int _id, String _name);//Constructor
- ○ public void setID(int x);//id mutator
- ○ public void setName(int x);//name mutator

- ○ public int getID();//id accessor
- ○ public String getName();//name accessor
- ○ public double getSalary();

To set the salary, write a method setSalary() which asks the user to enter the number of hours worked per week and hourly pay rate. Working 40 hours per week is the normal workload. However, if someone works more than 40 hours per week, it is considered overtime. The payment for overtime is 1.5 times more than the regular payment. In this method you have to compute the correct salary for an Employee. Moreover, for a manger, you have to add 10% in his salary to account for his management activities.

- ○ public void setSalary();//sets salary.

Create a test application EmployeeTest which creates an employee and a manager and computes their salary.

Task 4

Create a class called song which stores three attributes of a song: title of song (type String), its artist (type String), length of song in minutes (type int).

1. Your class should have a constructor that initializes the three instance variables.
2. Provide a set and a get method for each instance variable.

Create a test application songTest. In this application, create an array of songs. Fill the array by creating a new song with the title and artist and storing it in the appropriate position in the array. Print the contents of the array to the console. Your output should be as follows:

```
Yar Ko Ham Ne Ja-Bja Dekha by Abida Parveen (length 8min)
Allah ho Allah ho by Nusrat Fateh Ali Khan (length 5min)
.
.
.
.
```

In this test application, compute the total length of all the songs and display it on screen.