Chapter 3
Object Oriented Programming Concepts

1. Object Oriented Principles

Object-oriented programming languages use different concepts such as object, class, inheritance, polymorphism, abstraction, and encapsulation.

Classes and Objects

The concept of *classes* and *objects* is the heart of the OOP. A class is a framework that specifies what data and what methods will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. An object is a software bundle of related data and methods created from a class. For example we can think of the class **Student**. This class has some properties like name, address, marks, grade, and so on. Similarly it can have the methods like findPercentage(), findDivision(), etc. Now here we can see that from this class, we can create any number of objects of the type Student. In programming we say that an object is an instance of the class.

Encapsulation

Encapsulation is the process of combining the data and methods into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privilege to the data inside the class. So the term ***data hiding*** is possible due to the concept of encapsulation, since the data are hidden from the outside world.

Inheritance

Inheritance is the process of acquiring certain attributes and behaviors from parents. For examples, cars, trucks, buses, and motorcycles inherit all characteristics of vehicles. Object-oriented programming allows classes to inherit commonly used data and functions from other classes. If we derive a class from another class, some of the data and methods can be inherited so that we can reuse the already written and tested code in our program, simplifying our program.

Polymorphism

Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. However the behavior depends upon the attribute the name holds at particular moment. For example if we have the behavior called communicate for the objects vertebrates, then the communicate behavior applied to objects say dogs is quite different from the communicate behavior to objects human. So here the same name communicate is used in multiple process one for human communication, what we simply call talking. The other is used in communication among dogs, we may say barking, definitely not talking.

Abstraction

Abstraction is the essence of OOP. Abstraction means the representation of the essential features without providing the internal details and complexities. In OOP, abstraction is achieved by the help of class, where data and methods are combined to extract the essential features only. For example, if we represent chair as an object it is sufficient for us to

understand that the purpose of chair is to sit. So we can hide the details of construction of chair and things required to build the chair.

2. Defining Classes

A *class* specifies how objects are made. When you *construct* an object from a class, you are said to have created an *instance* of the class. As you have seen, all code that you write in Java is inside a class.

The standard Java library supplies several thousand classes for such diverse purposes. Nonetheless, in Java you still have to create your own classes to describe the objects of your application's problem domain.

A class is a framework that specifies what data/variables and what methods/functions will be included in objects of that class. It serves as a plan or blueprint from which individual objects are created. A class is also called user defined data type or programmers defined data type because we can define new data types according to our needs.

A class supports *encapsulation* is a key concept in working with objects. Formally, encapsulation is simply combining data and methods in one place. Using the concept of encapsulation, we can *hide data* from external use. The general syntax of the class definition is given below:

```
[Access][Modifiers] class ClassName [extends SuperClass] [implements Interface1 [, Interface2][, ……]]
{
        //body
}
```

Components:

- **Access:** public, private, protected etc. (discussed later).
- **Modifiers:** static, final etc. (discussed later).
- **Class name:** The class name, with the initial letter of each word capitalized by convention.
- **Superclass:** The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class (subclass) can only extend one parent (superclass). (discussed later).
- **Interfaces:** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface. (discussed later).
- **Body:** The class body generally contains variables and methods, surrounded by braces, {}.

2.1. Adding Variables

Data is encapsulated in a class by placing data fields (or variables) inside the body of the class definition. The general form of variable declaration is:

```
[Access] [Modifiers] data-type variableName;
```

Components:

- **Access:** public, private, protected etc. (discussed later).

- **Modifiers:** static, final etc. (discussed later).
- **Data type:** type of the variable. It may be primitive or user defined.
- **Variable name:** name of the variable. It is valid identifier with convention of starting with small letter, first word as a verb and the later words start with capital letter.

2.2. Adding Methods

To manipulate data contained in the class we add methods/functions inside the body of the class definition. The general form is:

```
[Access] [Modifiers] return-type methodName (ArgumentList) [exceptions list]
{
        //body
}
```

Components:

- **Access:** public, private, protected etc. (discussed later).
- **Modifiers:** static, final etc. (discussed later).
- **Return-type:** the data type of the value returned by the method or void if the method does not return a value.
- **Method name:** valid identifier with convention of starting with small letter, first word as a verb and the later words start with capital letter.
- **The parameter list in parenthesis:** a comma-separated list of input parameters, preceded by their data types, enclosed by parentheses. If there are no parameters, you must use empty parentheses.
- **Exception list:** List of exceptions (discussed later).
- **The method body, enclosed between braces:** the method's code, including the declaration of local variables, goes here.

2.3. A Simple Class

```
public class Rectangle
{
        private int length; //variable declaration
        private int breadth; //variable declaration
        public void setData(int l, int b) //method definition
        {
                length = l;
                breadth = b;
        }
        public int findArea() // method definition
        {
                return length * breadth;
        }
        public int findPerimeter() // method definition
        {
                return 2 * (length + breadth);
        }
}
```

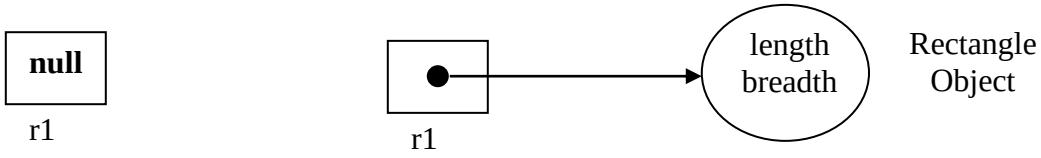
3. Creating Objects

Creating objects is also called object instantiation. Objects are created from the class using the **new** operator. The **new** operator creates an object of the specified class and returns a reference to that object. The main idea of using **new** is to create the memory that is required to hold an object of the particular type in run time i.e. to dynamically allocate the memory at the run time.

To create a usable object we must finish two steps: *declaring* the variable of its type and *instantiating* the object. The following example shows the creation of an object from the class Rectangle.

```
Rectangle r1; //declaring the variable of type Rectangle
r1 = new Rectangle(); //Instantiating an object
```

The execution of first statement creates the variable that holds reference of the class Rectangle using the name r1. It points nowhere (i.e. null) as shown below:



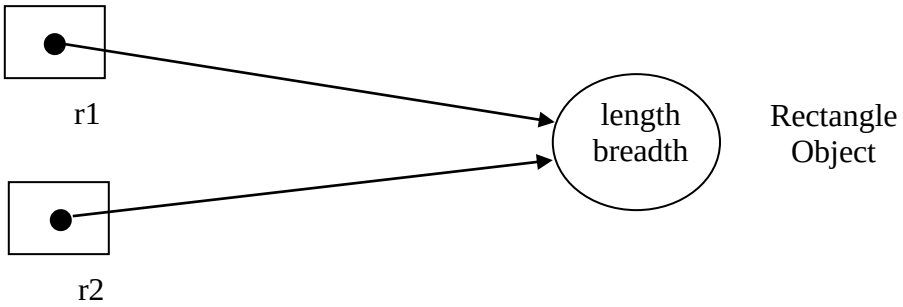
When the second statement is executed then the actual assignment of object reference to the variable is done as shown in above figure (right). The above two steps are equivalently written as:

```
Rectangle r1 = new Rectangle();
```

We can create more than one reference for same object and using different name we can manipulate same object. For example,

```
Rectangle r1 = new Rectangle();
Rectangle r2 = r1;
```

The pictorial representation of above code segment is as below:



Though r1 and r2 reference the same object they are not linked to each other so if we assign some other object to one of the name, previous link is removed from that name. For example, after some lines of code from above if we write

```
r1 = null;
```

The name r1 now does not point to the rectangle object.

4. Using Class Members

When an object of the class is created then the members are accessed using the '.' dot operator as shown below:

```
r1.setData(5, 2);
r1.findArea();
r1.findPerimeter();
```

Remember: Not all the members can be accessed from outside the class using the dot '.' operator. Here, we cannot access length and breadth from outside the class. (Why?)

5. A Complete Example

```
public class Rectangle
{
	private int length;
	private int breadth;
	public void setData(int l, int b)
	{
		length = l;
		breadth = b;
	}
	public int findArea()
	{
		return length * breadth;
	}
	public int findPerimeter()
	{
		return 2 * (length + breadth);
	}
}
public class MainRectangle
{
	public static void main(String[]args)
	{
		Rectangle r1 = new Rectangle();
		r1.setData(5, 2);
		System.out.println("Area = " + r1.findArea());
		System.out.println("Peremeter = " + r1.findPerimeter());
	}
}
```

6. Setters and Getters

We know that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods also called setters and getters. The **get** method returns the variable value, and the **set** method sets the value. The program below shows the use of set and gets methods.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case. The program below shows the use of set and gets methods.

```
public class Rectangle
{
	private int length;
	private int breadth;
	public void setLength(int l) //setter for length
	{
		length = l;
	}
	public int getLength() //getter for length
	{
		return length;
	}
	public void setBreadth(int b) //setter for breadth
	{
		breadth = b;
	}
	public int getBreadth() //getter for breadth
	{
		return breadth;
	}
	// other methods
}
```

Remember: If a method changes the value of variable(s), it is called **mutator method**. For example, set method in a class. The method that does not change but access value(s) is called **accessor** or **query method**. For example, get method in a class.

7. Constructors

Initializing the values of the variables of the object is very time-consuming process. Though we have set methods that can be used to assign values to the variables it is tedious too. So, to remedy this complexity we can take advantage of constructor for providing values to the instance variables.

The constructor initializes the variables of the object immediately after its creation and it will be simpler for us if we initialize variables while creating the object. Constructor has same name as of class and looks like a method except that it has no return type.

We have already seen the use of the constructor in above programs as Rectangle() [default constructor]. Though we have used the constructor we have not defined it and if we do not define the constructor the default constructor is called automatically. And if we define constructor ourselves then there will be no default constructor so in case of our need of default constructor, we have to create by ourselves. Since constructor looks like method, we can also provide arguments (parameters) to the constructor. The below program is the modification of the above program using constructors.

```
public class Rectangle
{
	private int length;
	private int breadth;
	public Rectangle() //default constructor
	{
	}
	public Rectangle(int l, int b) //parameterized constructor
	{
		length = l;
		breadth = b;
	}
	public void setData(int l, int b)
	{
		length = l;
		breadth = b;
	}
	public int findArea()
	{
		return length * breadth;
	}
	public int findPerimeter()
	{
		return 2 * (length + breadth);
	}
}
public class MainRectangle
{
	public static void main(String[]args)
	{
		Rectangle r1 = new Rectangle(7, 3);
		Rectangle r2 = new Rectangle();
		r2.setData(5, 2);
		System.out.println("Area of r1 = " + r1.findArea());
		System.out.println("Perimeter or r1 = " + r1.findPerimeter());
		System.out.println("Area of r2 = " + r2.findArea());
		System.out.println("Perimeter or r2 = " + r2.findPerimeter());
	}
}
```

8. Using this Keyword

The keyword **this** can be used within any method or constructor to represent the current object i.e. **this** is the reference to the object on which method is getting invoked. For example,

```
public class Rectangle
{
        private int length;
        private int breadth;
        public Rectangle(int length, int breadth)
        {
                this.length = length;
                this.breadth = breadth;
        }
        ……
        ……
}
```

If we look at the constructor definition above, we see the name **length** and **breadth** as both parameters and instance variables. Though normally in Java declaring two variables within the same scope is illegal it is permissible to have same name as parameter of a method and instance variables. In this case, we use **this** keyword to identify the instance variables.

9. Method Overloading

In Java, we can define different methods with the same name but with different parameters (either parameter type or number of parameters) within a class. This is one of the examples of **polymorphism**. If we define many methods with same name but with difference in parameters then this process is called **method overloading**. In this case the return type of the method does not matter and can be same as previously defined. For example,

```
public class Adder
{
        public int sum(int a, int b)
        {
                return a + b;
        }
        public int sum(int a, int b, int c)
        {
                return a + b + c;
        }
        public float sum(float a, int b)
        {
                return a + b;
        }
        public double sum(double a, double b)
        {
                return a + b;
        }
}
public class AdderTest
{
        public static void main(String[]args)
        {
```

```
                Adder a = new Adder();
                System.out.println(a.sum(5, 7));
                System.out.println(a.sum(5, 7, 2));
                System.out.println(a.sum(5.2f, 7));
                System.out.println(a.sum(5.3, 7.4));
        }
}
```

Remember: As we have already seen, constructor is much like method. So it is, most of the times, necessary to overload the constructor and this is possible with all the rules as defined for other methods. For example,

```
public class Rectangle
{
        public Rectangle()
        {
        }
        public Rectangle(int l, int b)
        {
                length = l;
                breadth = b;
        }
        //other methods and variables
}
```

10. Static Fields and Methods

10.1. Static Fields

If you define a field (variable or constant) as static, then there is only one such field per class. In contrast, each object has its own copy of non-static instance fields.

The value of static field is common to all objects and accessed without using a particular object if it is accessible. That is, the field belongs to the class as a whole rather than the objects created from the class. We use ***static*** modifier in the declaration of static fields. For example,

```
public class Circle
{
        private static final double PI = 3.1415; //static constant field
        private double radius; //non-static field
        public Circle(double r)
        {
                radius = r;
        }
        public double findArea()
        {
                return PI * radius * radius;
        }
        public double findCircum()
        {
```

```
            return PI * radius;
        }
}
```

10.2. Static Methods

Static methods are methods that do not operate on objects and are class-wide methods. These methods are declared using the static modifier and accessed without using a particular object if they are accessible. These methods are invoked with the class name like *ClassName.methodName(arguments)***.** When dealing with the static methods you must remember these facts:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access static fields and static methods directly.
- Static methods can access static variables and static methods directly.
- Static methods cannot access instance variables or instance methods directly--they must use an object reference. Also, static methods cannot use **this** and **super** in any way.

Example:

```
public class Calculate
{
        public static int cube(int x) // static method
        {
                return x * x * x;
        }
}
public class MainCalculate
{
        public static void main(String args[])
        {
                int result = Calculate.cube(5); //calling static method
                System.out.println(result);
        }
}
```

11. Parameters to Methods and Return Types

Information can be passed to methods as parameters. Methods may take parameters that are used inside the method. The number of parameters in the method is up to the nature of the method we are defining. For example, some method may not take any parameter, or it may take only one parameter, or two, and so on. In Java, method can be defined to take unknown number of parameters.

Java methods can take any type of data (primitive as well as objects) as parameters and return any type of value (primitive as well as objects). If the method takes parameter as primitive type then passing of argument from the caller is done as **call by value** and if the parameters are of object type the calling method is **call by reference**. The same process happens for the return type.

Call by value:

Here, the actual value of the argument (actual parameter) is passed to the method. In this case, actual parameter is copied to the formal parameter of the method so that changes made to the parameter inside the method are not reflected to the actual parameter.

Call by reference:

In this method, unlike the value, reference to an argument is passed to the formal parameter so that they refer to the same object location. In this process the modification to the parameter inside the method is reflected to the actual parameter.

Example:

```
public class Rectangle
{
        private int length;
        private int breadth;
        public Rectangle(int l, int b)
        {
                length = l;
                breadth = b;
        }
        public int getLength()
        {
                return length;
        }
        public int getBreadth()
        {
                return breadth;
        }
        public static void increment(int a)
        {
                a++;
        }
        public static void increment(Rectangle r)
        {
                r.length++;
                r.breadth++;
   }
}
public class MainProgram
{
        public static void main(String[]args)
        {
                int var = 7;
                Rectangle r1 = new Rectangle(10,5);
                Rectangle.increment(var);   //method call with primitive type parameter
                Rectangle.increment(r1);  //method call with object type parameter
                System.out.println("Var = " + var);
                System.out.println("Length = " + r1.getLength());
```

```
            System.out.println("Breadth = " + r1.getBreadth());
        }
}
```

12. Recursive Methods

Recursion is a process by which a `method` calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.
In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition. For example, the program below gives factorial of an integer number using recursive method.

```
public class RecursiveTest
{
        public static int factorial(int n)
        {
                if(n == 0)
                        return 1;
                else
                        return n * factorial(n - 1);
        }
}
public class MainRecursiveTest
{
        public static void main(String[]args)
        {
                int a = 7;
                System.out.println(a + "! = " + RecursiveTest.factorial(a));
        }
}
```

13. Access Control

Java has differently access modifiers namely **public**, **private**, and **protected**. Java also defines a default access level, also called package-private (no modifier). The protected modifier applies only when inheritance is involved.

Access modifiers for a class (top level [not nested]):
If we need our class to be visible to all other packages then we must declare it as public using “public” modifier. And if we need it to be visible in the same package only, we use no modifier. The use of other modifiers is not allowed. The term "visible" means you can use the accessible resources (call a method, read a field, whatever it is). Below we discuss both the cases:

```
package tu.pmc.bca;
public class Student //public specifier
{
```

```
        //class body
}
```

Class Student will be visible to all packages everywhere.

```
package tu.pmc.bca;
class Student //no specifier
{
        //class body
}
```

Class Student will be visible only within the same package (tu.pmc.bca), means same directory.

Access modifiers for members (methods, fields, and nested classes):

At member level we can use all the possible modifiers (private, public, and protected) and no modifier as well. Usage of the possible specifiers gives us four level of visibility. The below is the skeleton class describing all the possible specifiers.

```
package tu.pmc.bca;
public class Student
{
        int rollNumber;
        private String name;
        protected String subject;
        public String address;
        public int getRollNumber()
        {
                return rollNumber;
        }
        .....
        .....
}
```

If the member has no modifier then it is visible to all the classes within the package. This is often called package access. So, the *rollNumber* in the above skeleton is accessible within the classes inside *tu.pmc.bca* package.

If the member has **public** modifier then it is visible to all the classes in all the packages. So, *address* field and *getRollNumber* method in the above skeleton are accessible to all the packages.

If the member has **private** modifier then it is visible only inside class and nowhere else. So, the field *name* in the above skeleton is accessible to Students class only.

If the member has **protected** modifier then it is visible only inside the package and in the subclass outside the package. So, the field *subject* in the above skeleton is accessible to *tu.pmc.bca* package and to all the classes inheriting class Student. This modifier is used for the classes that can act as parent to open up visibility of their members to their child classes, but not to others.

Remember: If a class in a package is not public, then none of its members will be visible outside that package.

14. Nested and Inner Classes

It is possible to define a class within another class. Such classes are known as **nested classes**. They are same as any other class except that they are defined inside the body of another class. For example,

```
public class Outer
{
        //some code
        public class Inner
        {
                //some code
        }
}
```

Nested classes offer different benefits. Some of the reasons behind using nested classes are listed below:

- **Logical grouping of classes** – If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **Increased encapsulation** – Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **More readable, maintainable code** – Nesting small classes within top-level classes places the code closer to where it is used.

There are two types of nested classes: **static** and **non-static**. An **inner class** is a non-static nested class and it can be defined outside a method and inside a method.

14.1. Static Nested Classes

A static nested class cannot refer directly to instance variables or methods defined in its enclosing class; it can use them only through an object reference. However, it can access static fields and methods of the class directly. Instances of the static nested class can be created in any method of the outer class.

If the nested class is static, then they are not associated with instance of the outer class. Hence, we can create the instance of the static nested class in the classes other than outer and inner without creating the instance of outer class. For example,

```
public class Outer
{
        private static int outer_x = 100;
        public void test()
        {
                Inner in = new Inner();
                in.display();
        }
        public static class Inner
        {
                public void display()
```

```
                {
                        System.out.println(outer_x);
                }
        }
}
public class NestedMain
{
        public static void main(String[]args)
        {
                Outer out = new Outer();
                out.test();
                Outer.Inner in = new Outer.Inner();
                in.display();
        }
}
```

14.2. Inner Classes outside Methods

An inner class defined outside a method belongs to the class and has class scope (like other members). Like static nested classes, instances of the inner class can be created in any method of the outer class.

We can create the instance of the nested class in other classes than outer and inner by creating the instance of outer class. For example,

```
public class Outer
{
        private int outer_x = 100;
        public void test()
        {
                Inner in = new Inner();
                in.display();
        }
        public class Inner
        {
                public void display()
                {
                        System.out.println(outer_x);
                }
        }
}
public class NestedMain
{
        public static void main(String[]args)
        {
                Outer out = new Outer();
                out.test();
                Outer.Inner in = out.new Inner();
                in.display();
```

```
        }
}
```

Alternatively, we can create instance of the class Inner as follows:

```
Outer.Inner in = new Outer().new Inner();
```

14.3. Inner Classes inside Methods

An inner class defined inside a method belongs to the method and has local scope. The method can create instances of the inner class, but other methods of the outer class cannot see this inner class. For example,

```
public class Outer
{
        private int outer_x = 100;
        public void test()
        {
                class Inner
                {
                        public void display()
                        {
                                System.out.println(outer_x);
                        }
                }
                Inner in = new Inner();
                in.display();
        }
}
public class NestedMain
{
        public static void main(String[]args)
        {
                Outer out = new Outer();
                out.test();
        }
}
```

Inner classes defined inside a method have the following limitations:

- They cannot be declared with an access modifier
- They cannot be declared static
- Inner classes with local scope can only access variables of the enclosing method that are declared final, referring to a local variable of the method or an argument passed into the enclosing method

Inner classes can be defined inside a method without name (anonymous inner class), have following properties:

- Created with no name
- Defined inside a method
- Have no constructor
- Declared and constructed in the same statement

- Useful for event handling

15. Garbage Collection

We use **new** keyword to dynamically allocate the memory. Since the memory is created it must be destroyed to free the space if the object is not referenced by any variable. To do this task java takes a smarter approach that destroys the memory space used by the object whenever object is no longer in use. The process of freeing up memory used by objects that are not used is called garbage collection and this process is automatically done by java. It is not true that whenever the object is not used Java runs the garbage collector but the garbage collector is run occasionally. Normally, we do not have to take care about this thing in our programming. If you want to call the garbage collector by yourself then you can use the static method **gc** of the **System** class as:

System.gc();

Remember, the above request runs the garbage collector. It is not true that it forces the JVM to perform garbage collection.

finalize() method:

The **finalize()** method of **Object class** is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. This method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method has no parameter and has return type void and protected access modifier. For example,

```
public class GarbageTest
{
        protected void finalize()
        {
                System.out.println("Object destroyed!");
        }
        public static void main(String[]args)
        {
                GarbageTest g = new GarbageTest();
                g = null;
                System.gc();
        }
}
```

Output:

Object destroyed!

Exercises

1. Write a class Circle containing private variable radius of type float, suitable constructors, and two methods findArea and findCricumference to find area and circumference of circles respectively. Write a separate class MyCircle containing main method to create and use circle objects.

2. Write a class Rectangle containing a private variables length and breadth of type int, suitable constructors, and two methods findArea and findPerimeter to find area and perimeter of rectangles respectively. Write a separate class MyRectangle containing main method to create and use rectangle objects.
3. Write a class Box with private variables width, height and depth and methods to find volume and surface area. Use suitable constructors. Implement the class to find volume and surface area of two boxes.
4. Write a class Distance containing private variables feet of type int and inches of type float, suitable constructors, and two methods addDistance and compareDistance for adding and comparing two distance objects. Write a separate class MyDistance containing main method to create use distance objects.
5. Write a class Student with name, roll number, and marks in five subjects. Add two methods to calculate total and percentage of marks obtained. Use suitable constructors. Use this class to find percentage and division of five students.
6. Create a class called Time with three attributes hours, minutes, and seconds. Use appropriate constructors to initialize instance variables. Use methods to display the time in hh:mm:ss format and add two time objects. Implement the class to add and display time objects.
7. Create class SavingsAccount. Use a static variable annualInterestRate to store the annual interest rate for all account holders. Each object of the class contains a private instance variable savingsBalance indicating the amount the saver currently has on deposit. Provide method calculateMonthlyInterest to calculate the monthly interest by multiplying the savingsBalance by annualInterestRate divided by 12. This interest should be added to savingsBalance. Provide a static method modifyInterestRate that sets the annualInterestRate to a new value. Write a program to test class SavingsAccount. Instantiate two savingsAccount objects, saver1 and saver2, with balances of $2000.00 and $3000.00, respectively. Set annualInterestRate to 4%, then calculate the monthly interest and print the new balances for both savers. Then set the annualInterestRate to 5%, calculate the next month's interest and print the new balances for both savers.
8. Design a class to represent a bank account. Include the following members:
   Data members:
   - Name of the depositor
   - Account number
   - Type of account
   - Balance amount in the account

   Methods:
   - To assign initial values (constructor or method)
   - To deposit an amount
   - To withdraw an amount after checking balance
   - To display the name and balance
9. Write a program using recursive and iterative method to find factorial of a number.
10. Write a program using recursive and iterative method to find sum of first n natural numbers.
11. Write a program using recursive and iterative method to find n^{th} Fibonacci number.