Data Structures and Algorithms
Lab 2: Object-Oriented Programming - Part 1

I. Objective

After completing this tutorial, you can:

- Understand how to program an OOP program in Java,
- Understand *object* and *class* concepts,
- Understand *encapsulation* in OOP.

II. Java OOP

In this tutorial, we will focus on two basic concepts of Java OOP:

- Object (Section III),
- Class (Section IV).

III. Object

In real-world, we can find many **objects/entities** around us, e.g., chair, bike, dog, animal. All these objects have **state(s)** and **behavior(s)**. If we consider a *dog*, then its state is *name*, *breed*, *color*, and the behavior is *barking*, *wagging the tail*, *running*. That's it, an object has two characteristics[1]:

- **State**: represents data (value) of an object,
- **Behavior**: represents the behavior (functionality) of an object.

IV. Class

In the real world, you'll often find many individual objects all the same kind. There may be thousands of other bicycles in existence, all the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint/template from which individual objects are created.

To define a class in Java, the least you need to determine:

- Class's name – By convention, the first letter of a class's name is uppercased and subsequent characters are lowercased. If a name consists of multiple words, the first letter of each word is uppercased.
- Variables (State)
- Methods (Behaviors)
- Constructors

[1] Some documents define an object has three characteristics, i.e., *state*, *behavior*, and *identity*.

- Getter & setter (we'll discuss later in this tutorial)

1. Example program

Following is an example of a class:

- Name: Dog
- Properties: breed, age, color
- Methods: barking, hungry, sleeping

```
public class Dog
{
	String breed, color;
	int age;

	void barking()
	{
		System.out.println("barking...");
	}

	void sleeping()
	{
		System.out.println("sleeping...");
	}
}
```

A class can contain the following types of variables:

- Local variables – Variables defined inside methods, constructors or blocks are called local variables.
- Instance variables – Instance variables are variables within a class but outside any method.
- Class variables – Class variables are variables declared within a class, outside any method, with the ***static*** keyword.

A class can also have methods, e.g., *barking()*, *sleeping()*. Generally, method declarations have six components, in order:

- Modifiers – such as *public*, *private*, and *protected*.
- The return type – the data type of the value returned by the method, or void if the method does not return a value.
- The method's name.
- The parameter list in parenthesis – a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses. If there are no parameters, you must use empty parentheses.
- An exception list (to be discussed later).
- The method body, enclosed between braces – the method's code, including the declaration of local variables, goes here.

2. Constructor

Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class. Each time a new object is created, at least one constructor will be invoked.

A constructor must have the same name as the class. A class can have more than one constructor, but in most case, you need to define at least three types of constructor:

- Default constructor, with no parameter
- Parameterized constructor
- Copy constructor

The following program demonstrates how-to define constructors.

```
public class Dog
{
	String breed, color;
	int age;

	// Default constructor
	public Dog()
	{
		this.breed = "";
		this.color = "";
		this.age = 0;
	}

	// Parameterized constructor
	public Dog(String breed, String color, int age)
	{
		this.breed = breed;
		this.color = color;
		this.age = age;
	}

	// Copy constructor
	public Dog(Dog d)
	{
		this.breed = d.breed;
		this.color = d.color;
		this.age = d.age;
	}

	void barking()
	{
		System.out.println("barking...");
	}

	void sleeping()
	{
		System.out.println("sleeping...");
	}
}
```

In the above program, we use the ***"this"*** keyword to access instance variables. The ***"this"*** keyword is useful in case of parameterized constructor, we can clearly distinguish the instance variables and input parameters.

3. Java Access Modifiers

Java provides several access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels:

- Default – Visible to the package, no modifiers are needed.
- **Private** – Visible to the class only.
- **Public** – Visible to the world.
- **Protected** – Visible to the package and all subclasses (discuss later).

4. Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction (we'll discuss later).

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java:

- Declare the variables of a class as private/protected.
- Provide public *setter* and *getter* methods to modify and view the variables values.

The following program illustrates how-to achieve encapsulation in Java OOP program.

```java
public class Dog
{
	private String breed, color;
	private int age;

	// Default constructor
	public Dog()
	{
		this.breed = "";
		this.color = "";
		this.age = 0;
	}

	// Parameterized constructor
	public Dog(String breed, String color, int age)
	{
		this.breed = breed;
		this.color = color;
		this.age = age;
	}
```

```
        // Copy constructor
        public Dog(Dog d)
        {
                this.breed = d.breed;
                this.color = d.color;
                this.age = d.age;
        }

        void barking()
        {
                System.out.println("barking...");
        }

        void sleeping()
        {
                System.out.println("sleeping...");
        }

        public String getBreed()
        {
                return breed;
        }

        public void setBreed(String breed)
        {
                this.breed = breed;
        }

        public String getColor()
        {
                return color;
        }

        public void setColor(String color)
        {
                this.color = color;
        }

        public int getAge()
        {
                return age;
        }

        public void setAge(int age)
        {
                this.age = age;
        }
}
```

5. Test the class

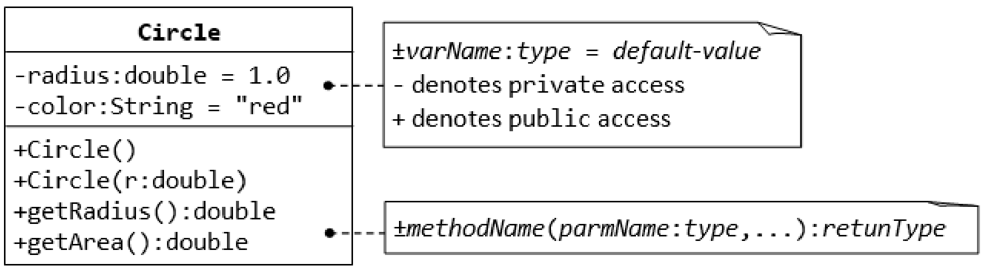
In order to test an implemented class, we need to define the ***main*** method, as follows.

```
public class DogTest
{

        public static void main(String[] args)
        {
                Dog dog = new Dog("Chihuahua", "brown", 4);
                System.out.println("Color: " + dog.getColor());
                System.out.println("Breed: " + dog.getBreed());
                System.out.println("Age: " + dog.getAge());

                dog.barking();
                dog.sleeping();
        }

}
```

V. Exercises

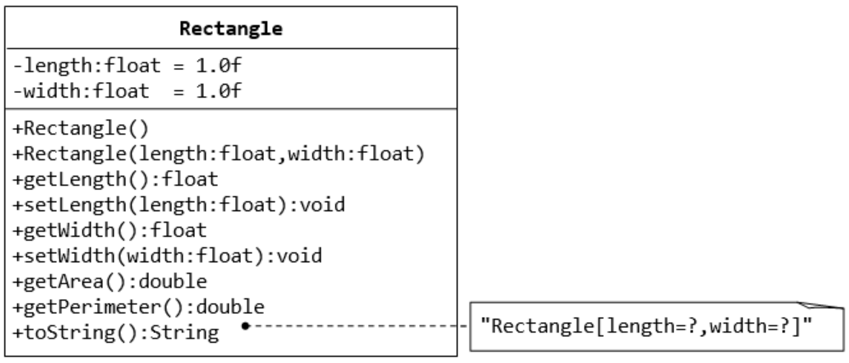
1. A class called **Circle** is designed as shown in the following class diagram. It contains:

- Two private instance variables: *radius* (of the type double) and *color* (of the type String), with default value of 1.0 and "red", respectively.
- Two overloaded constructors - a default constructor with no argument, and a constructor which takes a double argument for radius.
- Two public methods: *getRadius()* and *getArea()*, which return the radius and area of this instance, respectively.

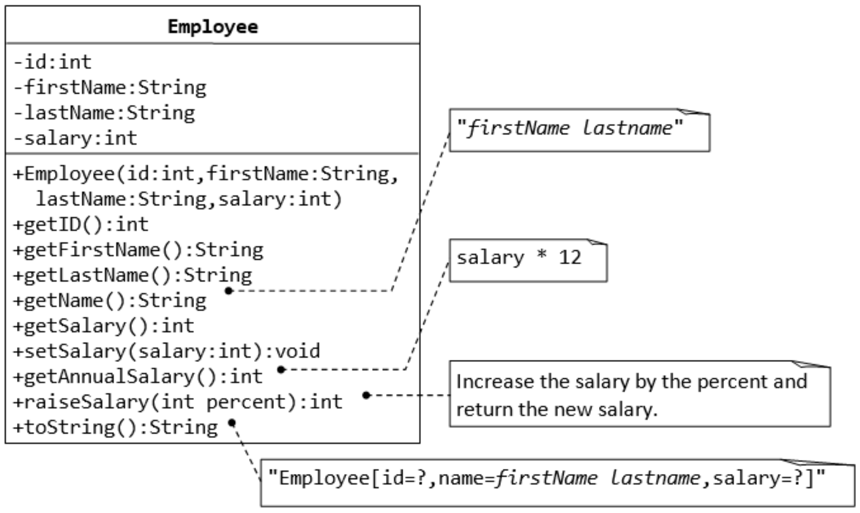


Implement the *Circle* class based on the definition.

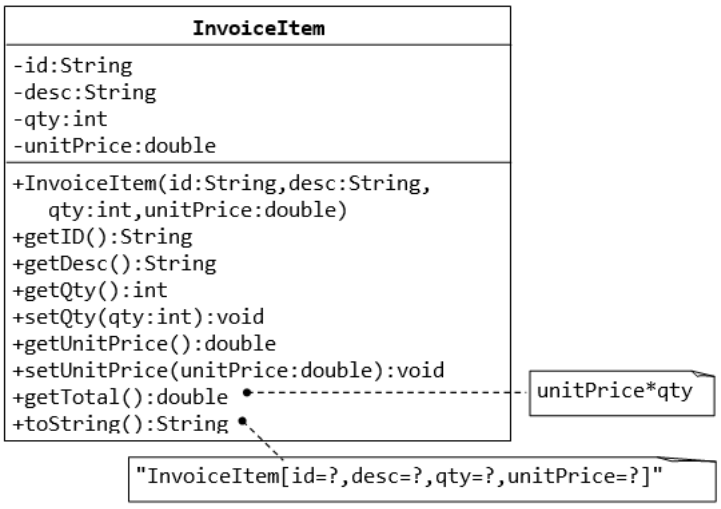
2. Implement the **Rectangle** class as defined as the following figure.



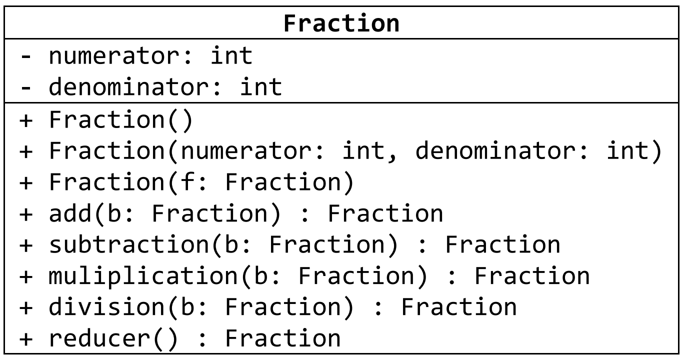
3. Implement the **Employee** class as defined as the following figure.


Employee
-id:int
-firstName:String
-lastName:String
-salary:int
+Employee(id:int,firstName:String, lastName:String,salary:int)
+getID():int
+getFirstName():String
+getLastName():String
+getName():String
+getSalary():int
+setSalary(salary:int):void
+getAnnualSalary():int
+raiseSalary(int percent):int
+toString():String
"firstName lastname"
salary * 12
Increase the salary by the percent and return the new salary.
"Employee[id=?,name=firstName lastname,salary=?]"

4. Implement the **InvoiceItem** class as defined as the following figure.



5. Implement the **Fraction** class as defined as the following figure.


Fraction
- numerator: int
- denominator: int
+ Fraction()
+ Fraction(numerator: int, denominator: int)
+ Fraction(f: Fraction)
+ add(b: Fraction) : Fraction
+ subtraction(b: Fraction) : Fraction
+ muliplication(b: Fraction) : Fraction
+ division(b: Fraction) : Fraction
+ reducer() : Fraction

VI. Homework

Read **Chapter 3**[2], in ***Data Structures & Problem Solving Using Java, 4th Edition (Mark A. Weiss)***.

[2] http://it.tdt.edu.vn/~dhphuc/teaching/cs501043/lab/objects-and-classes.pdf