OBJECT-ORIENTED PROGRAMMING
LAB 3: CLASS, OBJECT, ENCAPSULATION IN OOP

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 the real world, we can find many **objects/entities** around us, e.g., a chair, a bike, a dog, animals. All these objects have **state(s)** and **behavior(s).** If we consider a dog, then its states are name, breed, and color, and the behaviors are baking, wagging its tail, and running. That's it, an object has two characteristics.

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

IV. Class

In the real world, you will often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all with the same materials 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 a *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 uppercase, and subsequent characters are lowercase. If a name consists of multiple words, the first letter of each word is uppercase.
- Attributes/Properties (States)

- Methods (Behaviors)
- Constructors
- Getter & setter (we will discuss this later in this tutorial)

1. Example program

Following is an example of a class:

- Name: Student
- Attributes: name, gender, age
- Methods: studying, reading

```
public class Student {
    String name;
    String gender;
    int age;

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

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

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., `studying()`, `reading()`. 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 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 cases, you need to define at least three types of the constructor:

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

The following program demonstrates how to define constructors.

```
public class Student {
    String name;
    String gender;
    int age;

    public Student() {
        this.name = "";
        this.gender = "male";
        this.age = 0;
    }
    public Student(String name, String gender, int age) {
        this.name = name;
        this.gender = gender;
        this.age = age;
    }
    public Student(Student st) {
        this.name = st.name;
        this.gender = st.gender;
        this.age = st.age;
    }

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

In the above program, we use the "`this`" keyword to access instance variables. The "`this`" keyword is useful in the case of a 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 sub-classes (discuss later).

4. Overloading

Java supports the overloading method. It can distinguish between methods based on the method signatures. This means in the same class you can define two or more methods that have the same name if they have different parameter lists.

Overloading methods are differentiated by the number of arguments and the type of arguments passed into that method.

Example:

```
public class TestOverloading {
    public static int foo(int a, int b) {
        return a + b;
    }
    public static int foo(int a, int b, int c) {
        return a + b + c;
    }
    public static float foo(float a, float b) {
        return a + b;
    }
    public static float foo(float a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println(foo(5, 6));
        System.out.println(foo(5, 6, 7));
        System.out.println(foo(5.0f, 6.0f));
        System.out.println(foo(5.0f, 6));
    }
}
```

5. Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction (which we will 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 *getter* and *setter* methods to modify and view the variable's values.

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

```
public class Student {
    private String name;
    private String gender;
    private int age;

    public Student() {
        this.name = "";
        this.gender = "male";
        this.age = 0;
    }

    public Student(String name, String gender, int age) {
        this.name = name;
        this.gender = gender;
        this.age = age;
    }

    public Student(Student st) {
        this.name = st.name;
        this.gender = st.gender;
        this.age = st.age;
    }

    public void studying() {
        System.out.println("studying...");
    }

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

```
    public String getName() {
        return this.name;
    }

    public String getGender() {
        return this.gender;
    }

    public int getAge() {
        return this.age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

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

6. Test the class

To test an implemented class, we need to define the main method, as follows.

```
public class StudentTest {
    public static void main(String[] args) {
        Student student = new Student("Nguyen Van A", "male", 19);
        Student student1 = new Student();

        System.out.println("Name:" + student.getName());
        System.out.println("Gender:" + student.getGender());
        System.out.println("Age:" + student.getAge());

        student.studying();
        student.reading();

        System.out.println("Name:" + student1.getName());
        System.out.println("Gender:" + student1.getGender());
        System.out.println("Age:" + student1.getAge());
    }
}
```

7. Print the object

When you call method *System.out.println*() to print an object, it will call the *toString*() method from class *Object* to return a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object.

```
public class StudentTest {
    public static main(String[] args) {
        Student student = new Student("Nguyen Van A", "male", 19);

        System.out.println(student);
    }
}
```

To print the information of the object, you need to define the *toString*() method in your class. In the class Student above, let's define the *toString*() method and return the information of the student.

```
public class Student {
    …

    @Override
    public String toString() {
        return "Student[" + name + ", " + gender + ", " + age + "]";
    }
}
```

After defining this method, you can re-print the object student and observe the result.

```
public class StudentTest {
    public static main(String[] args) {
        Student student = new Student("Nguyen Van A", "male", 19);

        System.out.println(student);
    }
}
```

V. Exercises

1. A class called **Point2D** is designed as shown in the following class diagram. It contains:
   - Two private instance variables: **x** (of the type float) and **y** (of the type *float*), with default 0.0 and 0.0, respectively.
   - Two overloaded constructors: a default constructor with no argument, and a constructor that takes 2 float arguments for the **x** coordinate and **y** coordinate.

- Two public methods: *getX*() and *getY*(), which return the **x** coordinate and the **y** coordinate of this instance, respectively.

Point2D
- x: float = 0.0f - y: float = 0.0f
+ Point2D() + Point2D(x: float, y: float) + getX(): float + getY(): float

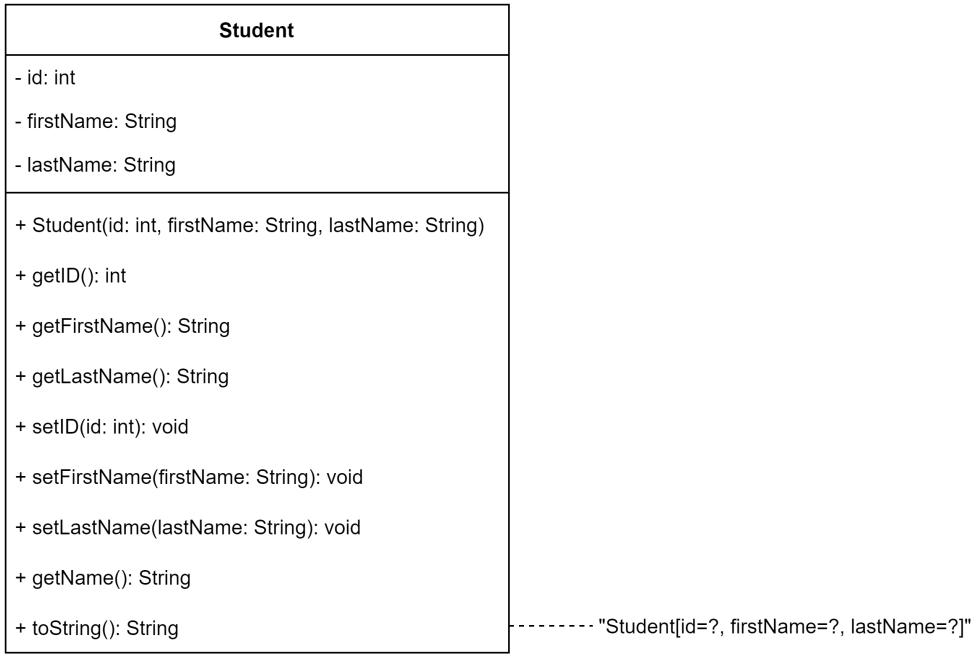
Implement a *Point2D* class based on the definition.

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

Rectangle
- width: float = 1.0f - length: float = 1.0f
+ Rectangle() + Rectangle(width: float, length: float) + getWidth(): float + getLength(): float + getArea(): float + getPerimeter(): float + setWidth(width: float): void + setLength(length: float): void + toString(): String

"Rectangle[width: float, length: float]"

3. Implement the **Student** class which is defined as the following figure.

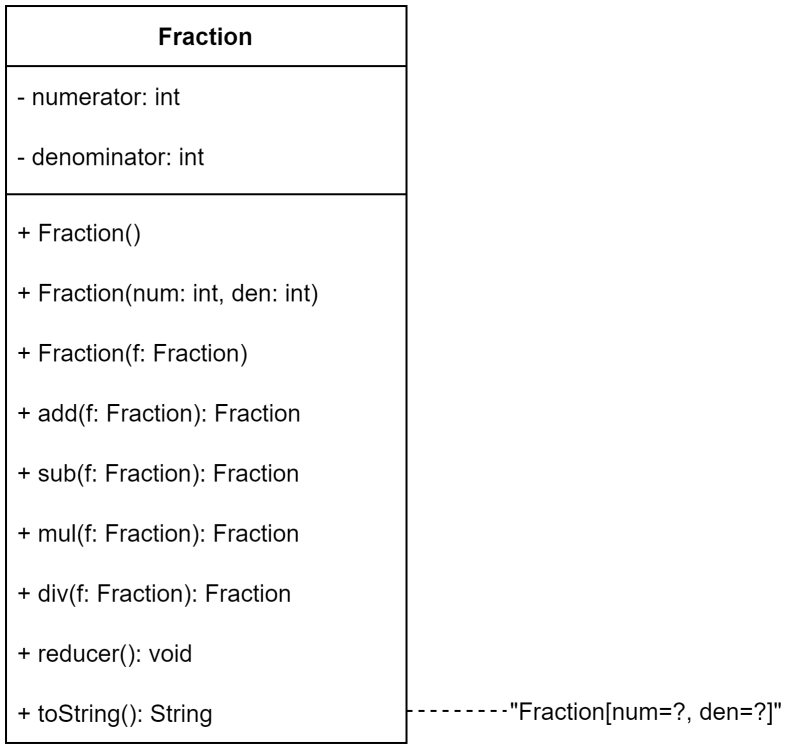


4. Define a House class according to the following description.
   - Attributes:
     - **houseCode**: String – the code of the house for sale (default: "A01")
     - **numOfBedRooms**: int – number of bedrooms (default: 2)
     - **hasSwimmingPool**: boolean (default: false)
     - **area**: double (default: 0)
     - **costPerSquareMeter**: double (default: 0)
   - Methods
     - **Constructor**: default constructor, fully parameterized constructor.
     - **Getter**, **setter** for all attributes.
     - **calculateSellingPrice():** double – calculate the selling price of the house according to the following formula:
       - *SubTotal* = area * costPerSquareMeter
       - If the house has a swimming pool, the subtotal will add 10% values of the subtotal.
         Selling *price* = SubTotal + 15% tax
       - Example: With the house having a swimming pool, *area* = 100, *costPerSquareMeter* = 2500000, the *subtotal* is 275000000, and the selling *price* is 316250000.

- **toString():** String – returns a string in the following format: "House[houseCode, numOfBedRooms, hasSwimmingPool, sellingPrice]"

Define a class with the main method to test the above class.

5. Implement the Fraction class which is defined as the following figure.



-- END --