

## OBJECT-ORIENTED PROGRAMMING

### LAB 1: JAVA, HOW TO PROGRAM

#### I. Objective

After completing this first lab tutorial, you can:

- Setup the Java Environment, compile and run the Java program.
- Know the primitive types, basic operators in Java.
- Use the basic statements: conditional statements, loop statements, method in Java.
- Use input and output syntax in Java.

#### II. Overview of Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

As of 2016, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

#### III. Environment Setup

This section will help you to set up the environment for the Java programming language. Following the steps below:

- Download and install the [Java SE Development Kit](#). You are recommended to use Java version 8. If you use another version, please be careful with the syntax. But all of your works when you are doing the exams or the exercises will be assigned with Java 8.
- Setting up the Path for Windows:
  - Right-click on My Computer and select Properties.
  - Click Advanced system settings.
  - Click the Path to include the directory path of installed JDK (as in FIG). Assuming you have installed in `C:\Program Files\Java\jdk1.8.0_281\bin`, then add it to the Path.

- Then, open the command prompt (CMD) window and type `java -version`. If you get the message that provides the information about the installed Java version, that means you have successfully set up the JDK and Java Path.

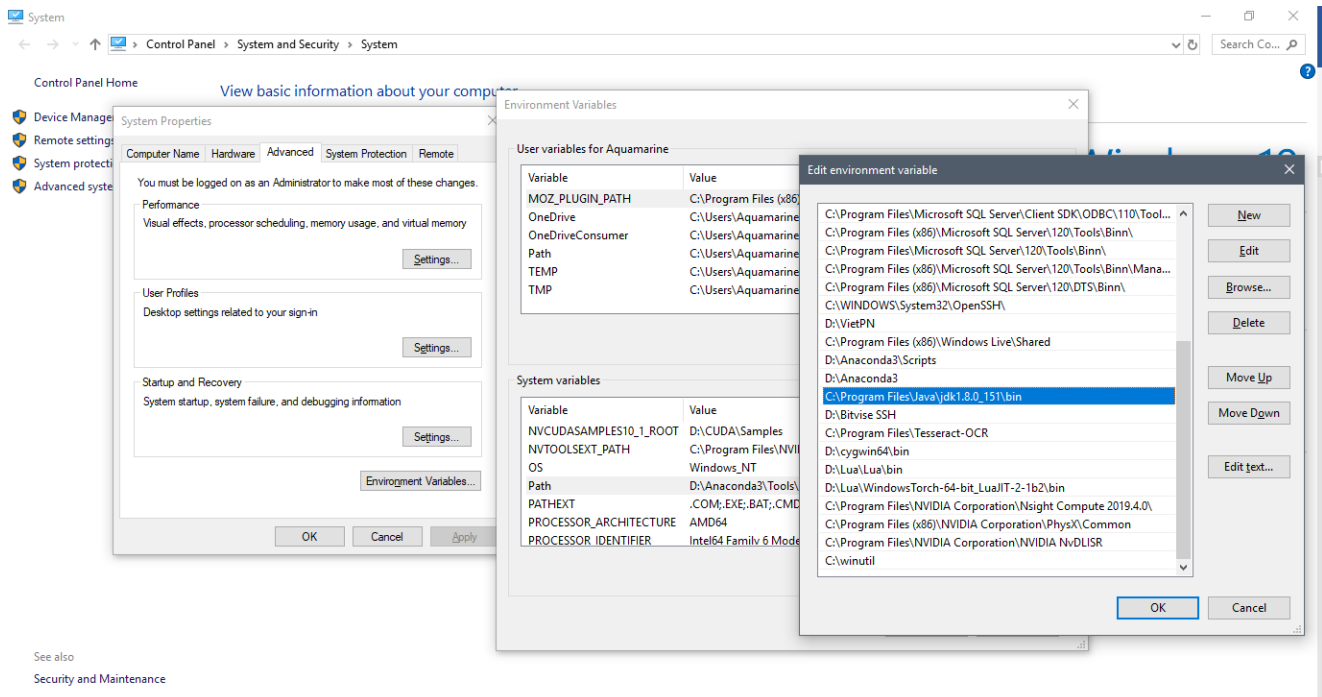


Figure 1. Environment Variables window

- For Java editor, we strongly recommend [Notepad++](#) for your very beginning course of Java. Beside that, you can use Visual Studio Code or Sublime Text. After completing this course, you can use other Java editors, e.g., Netbeans, Eclipse, IntelliJ IDEA.

## IV. First Java Program

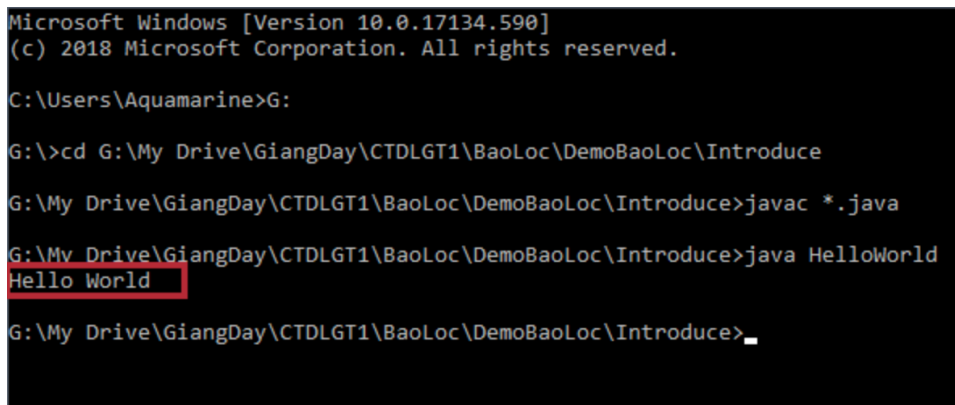
Let's look at and run your first Java program, which will print the "Hello World!" string on the screen.

```
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

Do the following steps to save the file, compile and run the program (as in Figure 2):

- Open the text editor program (Notepad++) and type the above code.
- Save as *MyFirstProgram.java*.

- Open the command prompt window (CMD) and navigate to the directory where you save the file.
- Type `javac MyFirstProgram.java` and press Enter to compile the source code.
- Type `java MyFirstProgram` to run the program.
- Now, you will see the "Hello World!" string on the screen.



```
Microsoft Windows [Version 10.0.17134.590]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Aquamarine>G:

G:\>cd G:\My Drive\GiangDay\CTDLGT1\BaoLoc\DemoBaoLoc\Introduce
G:\My Drive\GiangDay\CTDLGT1\BaoLoc\DemoBaoLoc\Introduce>javac *.java
G:\My Drive\GiangDay\CTDLGT1\BaoLoc\DemoBaoLoc\Introduce>java HelloWorld
Hello World
G:\My Drive\GiangDay\CTDLGT1\BaoLoc\DemoBaoLoc\Introduce>_
```

Figure 2. Compile and run Java program

## 1. Basic syntax

In java, you should keep in mind the following points:

- **Case Sensitivity:** Java is case sensitive, which means identifier Hello and hello would have a different meaning in Java.
- **Class Names:** For all class names, the first letter should be in Upper-Case. If several words are used to form the name of the class, each inner word's first letters should be in Upper-Case. Example: class `MyFirstProgram`
- **Method Names:** All method names should start with a Lower-Case letter. If several words are used to form the names of the method, then each inner word's first letter should be in Upper-Case. Example: `public void methodName()`
- **Program File Name:** Name of the program file has to exactly match the class name. Example: `class MyFirstProgram -> MyFirstProgram.java`
- `public static void main(String args[])`: Java program processing starts from the `main()` method which is a mandatory part of every Java program.

## 2. Java Identifiers

The identifier is the name used for classes, variables, and methods. To name an identifier in Java, you should remember the following points:

- All identifiers should begin with a letter (A to Z or a to z), currency character (\$) or an underscore (\_)
- After the first character, identifiers can have any combination of characters.
- A [keyword](#) cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.
- Examples of legal identifiers: `age`, `$salary`, `_value`, `__1_values`.
- Examples of illegal identifiers: `123abc`, `-salary`.

## 3. Java Modifiers

There are two categories of modifiers in Java:

- Access Modifier: `default`, `public`, `protected`, `private`.
- Non-access Modifiers: `final`, `abstract`, `strictfp`.

## 4. Comments in Java

Java supports both single-line and multiple-line comments, and they are similar to C and C++. All characters available inside any comments are ignored by the Java compiler.

```
// This is single-line comment
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
        /*
        This is multiple-line comment
        */
    }
}
```

## V. Data Types

In this first lab tutorial, we only focus on primitive data types. For the user-defined data types, we will discuss later in this course.

Type	Description	Default	Size
boolean	true or false	false	1 bit
byte	two complement in integer	0	8 bits
char	Unicode character	\u0000	16 bits
short	two complement in integer	0	16 bits
int	two complement in integer	0	32 bits
long	two complement in integer	0	64 bits
float	IEEE 754 floating point	0.0	32 bits
double	IEEE 754 floating point	0.0	64 bits

Let's look example below:

```
// MyFirstProgram.java
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        int a = 5, b = 10;
        int sum = a + b;
        System.out.println("a + b = " + sum);
    }
}
```

## VI. Basic Operators

Java provides a rich set of operators to manipulate variables, including:

- Arithmetic operators
- Relational operators
- Bitwise operators
- Logical operators
- Assignments operators
- Misc. operators

In this lab tutorial, we discuss arithmetic operators, relational operators, logical operators, and assignment operators. For the others, you can read in the textbook.

### 1. Arithmetic operators

Arithmetic operators are used in the mathematical expression in the same way that they are used in algebra.

Operators	Example
+ (addition)	a + b will give 30
- (subtraction)	a – b will give
* (multiplication)	a * b will give 200
/ (division)	a / b will give 2
% (modulus)	a % b will give 0
++ (increment)	b++ will give 21
-- (decrement)	b-- will give 19

### 2. Relational operators

The followings are the relational operations supported by Java language.

Operators	Example
== (equal to)	(a == b) is false
!= (not equal to)	(a != b) is true
> (greater than)	(a > b) is false
< (less than)	(a < b) is true
>= (greater than or equal to)	(a >= b) is false
<= (less than or equal to)	(a <= b) is true

### 3. Logical operators

The followings are the logical operators supported by Java language.

Operators	Example
&& (logical and)	(A && B) is false

(logical or)	(A    B) is true
! (logical not)	!(A && B) is true

#### 4. Assignment operators

Operators	Example
=	C = A + B will assign value of A + B into C
+=	C += A is equivalent to C = C + A
-=	C -= A is equivalent to C = C - A
*=	C *= A is equivalent to C = C * A
/=	C /= A is equivalent to C = C / A
%=	C %= A is equivalent to C = C % A

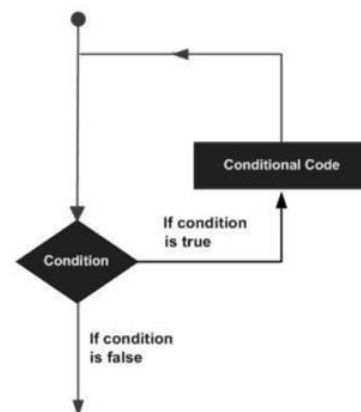
Let do some exercises to make your first Java program

#### VII. Loop Control

A loop statement allows us to execute a statement or group of statements multiple times (as in FIG).

Java programming language provides the following types of loop to handle looping requirements:

- **while** loop
- **for** loop, and enhanced for loop
- **do ... while** loop



## 1. *while* loop

A *while* loop statement in Java programming language repeatedly executes the target statement as long as a given condition is true.

Example:

```
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        int i = 1, sum = 0;
        {
            while (i <= 0 )
            {
                sum = sum + i;
                i++;
            }
            System.out.println("sum = " + sum);
        }
    }
}
```

## 2. *for* loop

A *for* loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. A *for* loop is useful when you know how many times a task will be repeated.

Example:

```
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        int sum = 0;
        {
            for (int i = 1; i <= 10; i++)
            {
                sum += i;
            }
            System.out.println("sum = " + sum);
        }
    }
}
```



### 3. *do ... while* loop

A *do ... while* loop is similar to a while loop, except that, *do ... while* loop is guaranteed to execute at least one time.

Example:

```
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        int i = 0, sum = 0;
        {
            do {
                sum = sum + i;
                i++;
            } while (i <= 10)
            System.out.println("sum = " + sum);
        }
    }
}
```

## VIII. Conditional Statements

Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Java provides two main types of conditional statements:

- **if ... else** statement
- **switch ... case** statement
- **? ... : ...** operator

Example 1:

```
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        int x = 11;
        if (x % 2 == 0)
        {
            System.out.println("x is even");
        }
        else
        {
            System.out.println("x is odd");
        }
    }
}
```

Example 2:

```
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        int x = 11, y = 12;
        if (x < y && x + y >= 10)
        {
            System.out.println("True");
        }
        else
        {
            System.out.println("False");
        }
    }
}
```

## IX. Methods

A Java method is a collection of statements that are grouped together to perform an operation. The syntax is as follows:

```
modifier returnType nameOfMethod (Parameter List)
{
    // statements
}
```

Example:

```
public class MyFirstProgram
{
    public static int findMax(int a, int b)
    {
        return (a > b)? a: b;
    }

    public static void main(String[] args)
    {
        int x = 5, y = 6;
        System.out.println("Max is " + findMax(x, y));
    }
}
```

According to the above example, *findMax* is the static method. So in the *main* function, you can call directly without creating a new object. You will learn more about the **method** in Lab 4.

## X. Text Output

Here is a list of the various print functions that we use to output statements:

- **print()**: This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.

Example:

```
class Demo {
    public static void main(String[] args)
    {
        System.out.print("Hello! ");
        System.out.print("Hello! ");
        System.out.print("Hello! ");
    }
}
```

Output:

```
Hello! Hello! Hello!
```

- **println()**: This method in Java is also used to display a text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

Example:

```
class Demo {  
    public static void main(String[] args)  
    {  
        System.out.println("Hello! ");  
        System.out.println("Hello! ");  
        System.out.println("Hello! ");  
    }  
}
```

Output:

```
Hello!  
Hello!  
Hello!
```

- **printf()**: This is the easiest of all methods as this is similar to printf in C. Note that System.out.print() and System.out.println() take a single argument, but printf() may take multiple arguments. This is used to format the output in Java.

Example:

```
class Demo {  
    public static void main(String args[])  
    {  
        int x = 100;  
        System.out.printf("Printing simple" + " integer: x = %d\n",x);  
  
        System.out.printf("Formatted with" + " precision: PI = %.2f\n", Math.PI);  
  
        float n = 5.2f;  
  
        System.out.printf("Formatted to " + "specific width: n = %.4f\n", n);  
  
        n = 2324435.3f;  
  
        System.out.printf("Formatted to " + "right margin: n = %20.4f\n", n);  
    }  
}
```

Output:

```
Printing simple integer: x = 100  
Formatted with precision: PI = 3.14  
Formatted to specific width: n = 5.2000  
Formatted to right margin: n =                2324435.2500
```

## XI. Text Input from Keyboard

Java provides a **Scanner** class, it is used to get user input, and it is found in the **java.util** package.

To use the Scanner class, create an object of the class and use any of the available methods found in the documentation.

Method	Description
nextBoolean()	Reads a boolean value from user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Read a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

Example:

```
import java.util.Scanner;

class MyFirstProgram
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        String name = sc.nextLine();
        int age = sc.nextInt();

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

## XII. Exercises

1. Write a program to print your name, date of birth, and student ID.
2. Write a program to compute the area of a rectangle with a height and base provided by user.  
( $area = \frac{1}{2} \times base \times height$ )
3. Write a function to convert the temperature from Fahrenheit to Celsius and a function to convert the temperature from Celsius to Fahrenheit.
4. Write a function to check whether a year is a leap year or not.
5. Write a function to find the minimum between three numbers.
6. Write a function to check whether a number is even or odd.
7. Write a function that receive a character and check whether it is alphanumeric or not.
8. Write functions to calculate the below formulas with  $n$  is provided by user (each formula is one function):
  - a.  $S = 1 + 2 + 3 + \dots + n$
  - b.  $P = 1 \times 2 \times 3 \times \dots \times n$
  - c.  $S = 1 + 2^1 + 2^2 + \dots + 2^n$
  - d.  $S = \frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \dots + \frac{1}{n}$
  - e.  $\sum_{i=1}^n i^2$
9. Write a function to find the first digit and a function to find last digit of a number.
10. Write a function to calculate sum of digits and a function to calculate product of digits of a number.
11. Write a function to return the remainder of a division.
12. Write a function to count number of digits in a number.
13. Write a function to reverse the input integer number.
14. Write a function to check whether a number is palindrome or not.
15. \*Write a program to simulate a vending machine. This is the output of the demo program, you can use this output to build your program (*Hint: you should use **while** statement and **switch...case...** statement*):

```
PS D:\Work\Teaching\HK2.2021\LTHDT\TestBuildLab\Lab01\VendingMachine> java VendingMachine
----Menu----
1. Coca
2. Pepsi
3. Sprite
4. Snack
5. Shutdown Machine
Please enter the number:
1
The price of Coca is: 2$, please enter the amount of money:
5
Your change is 3.0$.
----Menu----
1. Coca
2. Pepsi
3. Sprite
4. Snack
5. Shutdown Machine
Please enter the number:
1
The price of Coca is: 2$, please enter the amount of money:
1
Not enough money to buy this item. Please select again.
----Menu----
1. Coca
2. Pepsi
3. Sprite
4. Snack
5. Shutdown Machine
Please enter the number:
6
Please enter the valid number.
----Menu----
1. Coca
2. Pepsi
3. Sprite
4. Snack
5. Shutdown Machine
Please enter the number:
5
Machine is shutting down.
PS D:\Work\Teaching\HK2.2021\LTHDT\TestBuildLab\Lab01\VendingMachine>
```

-- THE END --

## OBJECT-ORIENTED PROGRAMMING LAB 2: JAVA, HOW TO PROGRAM (CONT.)

### I. Objective

In this second tutorial, you will:

- Practice with array in Java.
- Have basic knowledge about object, how to create object in Java and practice it with the sample class which defined by Java.

### II. Array

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. To declare an array, we have 4 possible ways:

```
dataType[] arrayName;  
dataType arrayName[];  
dataType[] arrayName = new dataType[arraySize];  
dataType[] arrayName = {value0, value1, ..., valueK};
```

Example:

```
public class MyFirstProgram  
{  
    public static void main(String[] args)  
    {  
        int[] a = {1, 2, 3, 4, 5};  
        int sum1 = 0, sum2 = 0;  
        for (int i = 0; i < a.length; i++)  
        {  
            sum1 = sum1 + a[i];  
        }  
        System.out.println("sum1 = " + sum1);  
        for (int x : a)  
        {  
            sum2 = sum2 + x;  
        }  
        System.out.println("sum2 = " + sum2);  
    }  
}
```

We have a special for loop in the above sample called *enhanced for* loop. The *enhanced for* loop is mainly used to traverse a collection of elements including arrays. The syntax is as follows.

Example:



```
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        int[] a = {1, 3, 5, 7, 9};
        for (int x : a)
        {
            System.out.println(x);
        }
    }
}
```

### III. Classes and objects in Java

Java is an Object-Oriented Programming (OOP) language. Everything you work in Java is through classes and objects. In this lab, we just learn how to create an object from the available class in Java, we will learn about this topic more carefully in Lab 4.

Integer is a class, which is defined in the **java.lang.Integer** package. This class wraps a value of the primitive type *int* in an object. You can read more about this class in [here](#).

#### Object is an instance of a Class.

Example:

```
class Test{
    public static void main(String[] args){
        Integer num = new Integer(3);

        Integer num1 = new Integer(5);

        Integer x = num;

        Integer y;

        System.out.println(num);
        System.out.println(num1);
        System.out.println(x);
        //System.out.println(y);
    }
}
```

Output:

```
3
5
3
```

With the above example, the variables: *num*, *num1*, *x*, *y* created from the Integer class. In other words, the variables: *num*, *num1*, *x*, *y* are the pointers to the objects. But with variable *y*, it hasn't been

initialized, so its reference points to **null**. With **null**, you can't do anything and it may cause **NullPointerException** when you try to use it.

The format of code when you want to create an object from the class is:

```
ClassName instanceName = new ClassConstructor([parameter1, parameter2, ...])
```

Let observe another example:

```
class Test1{
    public static void main(String[] args){
        Integer x = new Integer(3);
        Integer y = new Integer(3);

        System.out.println(x == y);
        System.out.println(x.equals(y));
    }
}
```

Output:

```
false
true
```

With objects, we don't use operator "==" to compare values, because this operator will compare the addresses and there are the pointers so they have different address. Integer supports the method called **equals** to compare the value of two Integer objects.

Next, we will learn how to invoke the **non-static method** and **static method** from a class. To invoke a static method, you do not need to create an object, instead you can invoke directly from class name. Contrarily, with non-static method, you need to create an object to invoke it.

Example:

```
class Test2{
    public static void main(String[] args){
        Integer x = new Integer(3);

        int y = x.intValue(); //intValue() is a non-static method of Integer class

        int z = Integer.sum(2,3); //sum() is a static method of Integer class

        System.out.println(y);
        System.out.println(z);
    }
}
```

Output:

```
3
5
```

#### **IV. Exercises**

1. Write a function *public static int findMax(int arr[])* to find the maximum value of an array.
2. Write a function to find the minimum value of an array.
3. Write a function to sum all even numbers of an array.
4. Write a function to count how many specific element in an array.
5. Write a function to count how many prime number in an array.
6. Write a function *public static int find(int arr[], int k)* to find the index of an element *k* in an array, if the element doesn't exist in an array return -1. (the first element index is 0)
7. Write a function *public static void square(int arr[])* to square all elements of an array.
8. Write a function *public static Integer findMax(Integer []arr)* to find the maximum value of an Integer object array.
9. \*Write a function *public static int[] divisibleNumbers(int arr[], int k)* to find the elements which divisible by *k* in an array. (Ex: *a = [1,2,3,4,5,6,7]* with *k = 2*  $\rightarrow$  *[2,4,6]*)
10. \*Write a function to find the third largest element in an array.

## OBJECT-ORIENTED PROGRAMMING

### LAB 3: STRINGBUFFER, STRINGBUILDER, STRINGTOKENIZER

#### I. Objective

After completing this tutorial, you can:

- Understand how to program with **StringBuffer**, **StringBuilder**, and **StringTokenizer**.

#### II. The class *StringBuffer*

In some situations, it is useful to be able to alter the sequence of characters stored in a string. But class **String** supports only nonmutable string. To create mutable string use the class **StringBuffer** from the package **java.lang**. This class provides the same functionality as the class **String**, plus the following methods that actually modify the value stored in the **StringBuffer** object.

##### StringBuffer Constructors

###### 1. **StringBuffer()**

Creates a **StringBuffer** with empty content and 16 reserved characters by default.

```
StringBuffer sb = new StringBuffer();
```

###### 2. **StringBuffer(int sizeOfBuffer)**

Creates a **StringBuffer** with the passed argument as the size of the empty buffer.

```
StringBuffer sb = new StringBuffer(20);
```

###### 3. **StringBuffer(String string)**

Creates a **StringBuffer** with the passed **String** as the initial content of the buffer. 16 contingent memory characters are pre-allocated, not including the buffer, for modification purposes.

```
StringBuffer sb = new StringBuffer("Hello World!");
```

##### StringBuffer Methods

###### 1. **length()**

Returns the **StringBuffer** object's length.

```
StringBuffer sb = new StringBuffer("Hello");  
int sbLength = sb.length();  
System.out.println("String Length of " + sb + " is " + sbLength);  
// String Length of Hello is 5
```

## 2. capacity()

Returns the capacity of the **StringBuffer** object.

```
StringBuffer sb = new StringBuffer("Hello");  
int sbCapacity = sb.capacity();  
System.out.println("Capacity of " + sb + " is " + sbCapacity);  
// Capacity of Hello is 21
```

## 3. append()

Appends the specified argument string representation at the end of the existing String Buffer. This method is overloaded for all the primitive data types and Object.

```
StringBuffer sb = new StringBuffer("Happy ");  
sb.append("New Year ");  
sb.append(2020);  
System.out.println(sb);  
// Happy New Year 2020
```

## 4. insert()

The string **str** is inserted into this string buffer at the index indicated by offset. Any characters originally above that position are moved up and the length of this string buffer increased by the length of **str**. If **str** is null, the string null is inserted into this string buffer.

```
StringBuffer sb = new StringBuffer("HelloWorld ");  
sb.insert(5, " ");  
sb.insert(sb.length(), 2020);  
System.out.println(sb);  
//Hello World 2020
```

## 5. reverse()

Reverses the existing String or character sequence content in the buffer and returns it. The object must have an existing content or else a **NullPointerException** is thrown.

```
StringBuffer sb = new StringBuffer("Hello World");  
System.out.println(sb.reverse());  
//dlroW olleH
```

## 6. delete(int start, int end)

Removes the characters in a substring of this string buffer starting at index **start** and extending to the character at index **end - 1** or to the end of the string buffer if no such character exists. If **start** is equal to **end**, no changes are made. This method may throw **StringIndexOutOfBoundsException** if the value of **start** is negative, greater than the length of the string buffer, or greater than **end**.

```
StringBuffer sb = new StringBuffer("Hello World");  
System.out.println(sb.delete(5,11));  
//Hello
```

### 7. setCharAt(int index, char ch)

The character at index **index** of this string buffer is set to **ch**. This method may throw **IndexOutOfBoundsException** if the value of index is negative or is greater than or equal to the length of the string buffer.

```
StringBuffer sb = new StringBuffer("Hello World");  
System.out.println(sb.setCharAt(2, 'E'));  
//Hello World
```

### 8. replace(int start, int end, String str)

Replaces the characters in a substring of this string buffer with characters in the specified string **str**. The substring to be replaced begins at index **start** and extends to the character at index **end - 1** or to the end of the string buffer if no such character exists. The substring is removed from the string buffer, and then the string **str** is inserted at index **start**. If necessary, the string buffer is lengthened to accommodate the string **str**. This method may throw **StringIndexOutOfBoundsException** if the value of start is negative, greater than the length of the string buffer, or greater than **end**.

```
StringBuffer sb = new StringBuffer("Hello World!");  
System.out.println(sb.replace(6,11,"Earth"));  
//Hello Earth!
```

## III. The class StringBuilder

Java **StringBuilder** class is mutable sequence of characters. **StringBuilder** Class can be comparable to String however the **StringBuilder** class provides more versatility because of its modification features.

- **StringBuilder** class provides an API similar to **StringBuffer**, but unlike **StringBuffer**, it doesn't guarantee thread safety.
- Java **StringBuilder** class is designed for use as a drop-in replacement for **StringBuffer** in places where the string buffer was being used by a single thread (as is generally the case).
- If execution speed and performance is a factor, **StringBuilder** class can be used in place of **StringBuffer**.
- The bread-and-butter operations provided by the **StringBuilder** Class are the **append()** and **insert()** methods. These methods are overloaded within **StringBuilder** in order to accommodate different data type.

- The general process flow of **StringBuilder** append and insert methods is: (1) converts a given data to a string then (2) appends or inserts the characters of that string to the string builder. Java **StringBuilder** **append()** method always adds these characters at the end of the builder; **insert()** method inserts character(s) at a specified point.

## StringBuffer and StringBuilder

StringBuffer	StringBuilder
Synchronized, hence thread safe.	Not synchronized, not thread safe.
Operates slower due to thread safety feature	Better performance compared to StringBuffer
Has some extra methods – substring, length, capacity etc.	Not needed because these methods are present in String too

## StringBuilder Constructors

### 1. StringBuilder()

Creates an empty string builder with a default capacity of 16 (16 empty elements).

```
StringBuilder str = new StringBuilder();
```

### 2. StringBuilder(CharSequence seq)

Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements trailing the CharSequence.

```
StringBuilder str1 = new StringBuilder("AAAABBBCCCC");
```

### 3. StringBuilder(int capacity)

Creates an empty string builder with the specified initial capacity.

```
StringBuilder str2 = new StringBuilder(10);
```

### 4. StringBuilder(String str)

Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

```
StringBuilder str3 = new StringBuilder("Hello World");
```

## StringBuilder Methods

### 1. append()

**StringBuilder** **append()** method concatenates or attaches the passed String argument with the existing declared string. It attaches it after the declared string.

```
StringBuilder sb = new StringBuilder("Hello ");  
sb.append("World");  
System.out.println(sb);  
// Hello World
```

### 2. insert()

**StringBuilder** **insert()** method inserts the passed String argument at the passed **String** index.

```
StringBuilder sb = new StringBuilder("HellWorld");  
sb.insert(4, "o ");  
System.out.println(sb);  
// Hello World
```

### 3. replace(int startIndex, int endIndex, String str)

**StringBuilder** **replace()** method replaces the existing declared string. String replacement occurs from the passed **startingIndex** up to the **endingIndex**.

```
StringBuilder sb = new StringBuilder("Hello World!");  
sb.replace(6,11,"Earth");  
System.out.println(sb);  
// Hello Earth!
```

### 4. delete(int startIndex, int endIndex)

**StringBuilder** **delete()** method deletes a character or sets of characters. Deletion occurs at passed **startingIndex** up to **endingIndex**.

```
StringBuilder sb = new StringBuilder("Dung Cam Quang");  
sb.delete(0,9);  
System.out.println(sb);  
//Quang
```

### 5. reverse()

The **reverse()** method of **StringBuilder** class reverses the existing declared string. Invoking a **reverse()** method on a **StringBuilder** object with no existing declared value throws **NullPointerException**.



```
StringBuilder sb = new StringBuilder("devil");  
sb.reverse();  
System.out.println(sb);  
// lived
```

## 6. capacity()

The `capacity()` method of **StringBuilder** class returns the current capacity of the **StringBuilder** object. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by  $(old\_capacity \times 2) + 2$  e.g. at current capacity 16, it becomes  $(16 \times 2) + 2 = 34$ .

```
StringBuilder sb=new StringBuilder();  
System.out.println(sb.capacity()); // default value 16  
sb.append("Java");  
System.out.println(sb.capacity()); // still 16  
sb.append("Hello StringBuilder Class!");  
System.out.println(sb.capacity()); // (16*2)+2
```

## IV. The class StringTokenizer

Another useful class when working with strings is **StringTokenizer** in the package `java.util`. This class allows a program to break a string into pieces or tokens. The tokens are separated by characters known as delimiters. When you create a **StringTokenizer** instance, you must specify the string to be tokenized. Other constructors within **StringTokenizer** allow you to specify the delimiting characters and whether the delimiting characters themselves should be returned as tokens.

### String Tokenizer Constructors

#### 1. StringTokenizer(String str)

This constructor creates a string tokenizer for the specified string `str`. The tokenizer uses the default delimiter set, which is the space character, the tab character, the newline character, the carriage-return character, and the form feed character. Delimiter characters themselves are not treated as tokens.

```
StringTokenizer st1 = new StringTokenizer("Hello guy How are you");
```

#### 2. StringTokenizer(String str, String delim)

This constructor creates a string tokenizer for the specified string `str`. All characters in the `delim` string are the delimiters for separating tokens. Delimiter characters themselves are not treated as tokens.

```
StringTokenizer st2 = new StringTokenizer("JAVA : Code : String", " :");
```

### 3. StringTokenizer(String str, String delim, boolean returnTokens)

This constructor creates a string tokenizer for the specified string **str**. All characters in the **delim** string are the delimiters for separating tokens. If the **returnTokens** flag is true, the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length 1. If the flag is false, the delimiter characters are skipped and serve only as separators between tokens.

```
StringTokenizer st3 = new StringTokenizer("JAVA : Code : String", " :", true);
```

#### StringTokenizer Methods

##### 1. nextToken()

Returns the next token in the string. If there are no more tokens in the string, it throws the exception **NoSuchElementException**.

##### 2. hasMoreTokens()

Returns true if the string contains more tokens.

```
StringTokenizer st1 = new StringTokenizer("Hello guy How are you", " ");  
while (st1.hasMoreTokens())  
{  
    System.out.println(st1.nextToken());  
}  
// Hello  
// guy  
// How  
// are  
// you
```

##### 3. countTokens()

Calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception. The current position is not advanced. This function returns the number of tokens remaining in the string using the current delimiter set.

#### IV. Exercises

1. Write a Java program:
  - Split the full name into first name and last name (using StringTokenizer).
2. Write a Java program:
  - Count number of words in string (using StringTokenizer)
  - Concatenate one string contents to another (using StringBuffer or StringBuilder).
  - Check a string is palindrome or not (using StringBuffer or StringBuilder).
3. Write a Java program:
  - Remove leading white spaces from a string (using StringTokenizer and StringBuffer/StringBuilder).
  - Remove trailing white spaces from a string (using StringTokenizer and StringBuffer/StringBuilder).
  - Remove extra spaces from a string (using StringTokenizer and StringBuffer/StringBuilder).
4. Write a Java program:
  - Write a function that processes a String paragraph, count occurrences of each word in the paragraph, and store them in a 2D-array.
  - Write a main function, use the above function and print the matrix to the screen.

For example: "I live in Ho Chi Minh city. I study at TDTU"

Screen:

```
I 2
live 1
in 1
Ho 1
Chi 1
Minh 1
city 1
study 1
at 1
TDTU 1
```

## OBJECT-ORIENTED PROGRAMMING

### LAB 4: 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. 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 its tail, 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 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.
- **Variables (State)**

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

### 1. Example program

Following is an example of a class:

- Name: Student
- Properties: 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 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 defined 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 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 sub-classes (discuss later).

### 4. Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction (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 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;
    }

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

    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;
}
}
```

## 5. Test the class

In order 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());
    }
}
```

## 6. 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 toString() method in your class. In the class Student above, let define 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 Point 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 **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

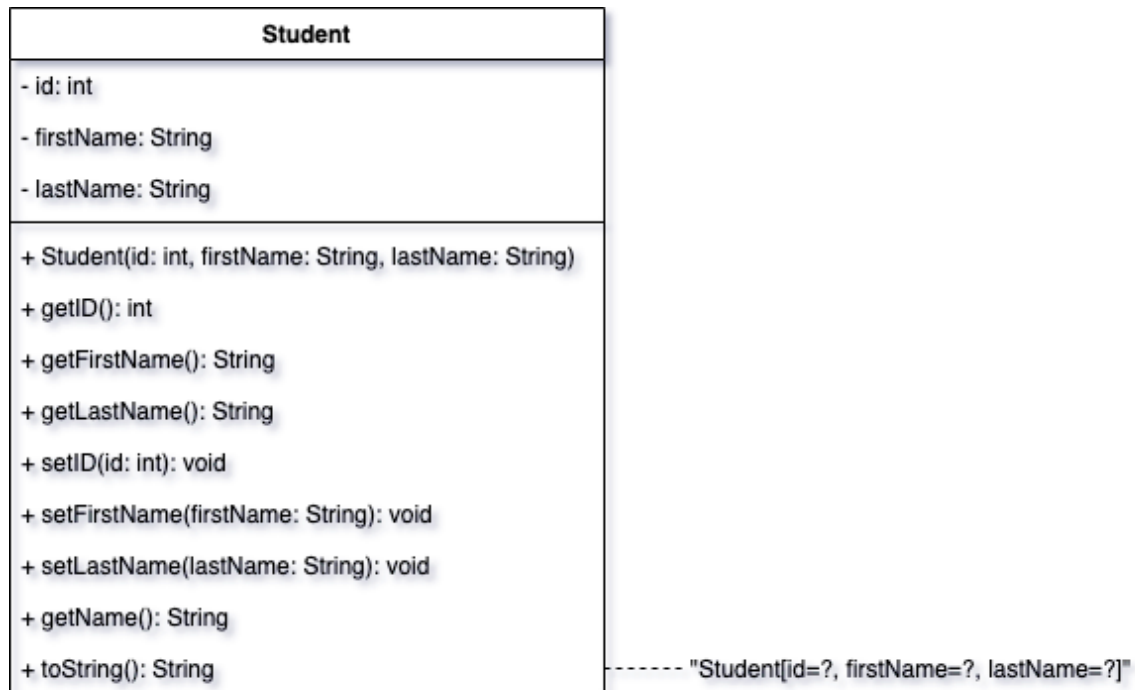
Implement Point class based on the definition.

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

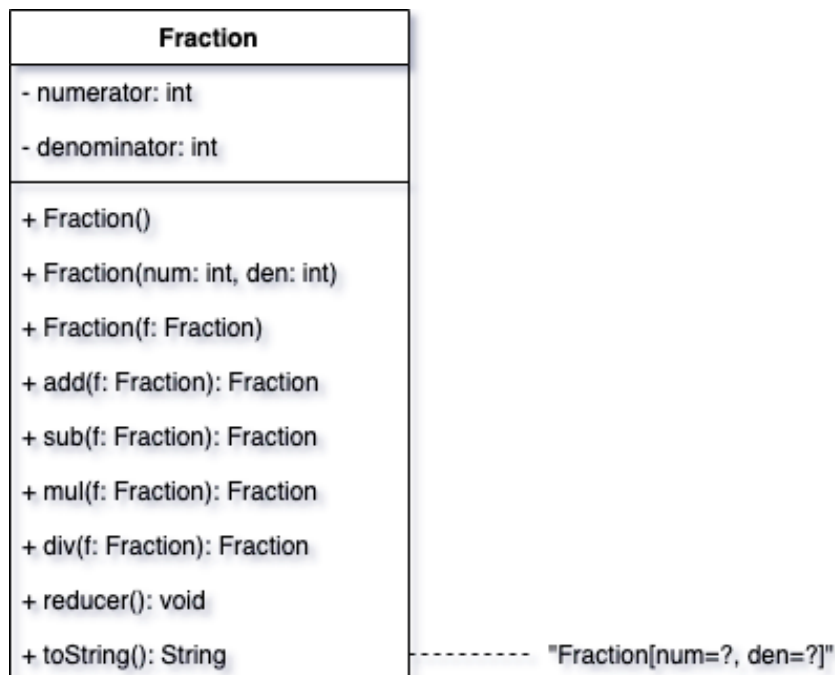
Rectangle
- width: float = 1.0f - length: float = 1.0f
+ Rectangle() + Rectangle(width: float, length: float) + getWidth(): float + getLength(): float + setWidth(width: float): void + setLength(length: float): void + toString(): String

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

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



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



## OBJECT-ORIENTED PROGRAMMING

### LAB 6: INHERITANCE

#### I. Objective

After completing this tutorial, you can:

- Understand *inheritance* in OOP.

#### II. Definition

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

The class which inherits the properties of other class is known as subclass (derived class, child class) and class whose properties are inherited is known as superclass (base class, parent class).

##### 1. extends keyword

The extends keyword is used to inherit the variables and methods of a class (except private variables and private methods).

```
public class Super
{
    //...
}

public class Sub extends Super
{
    //...
}
```

##### 2. super keyword

The super keyword in java is a reference variable which refers to its parent class object. The usage of super keyword:

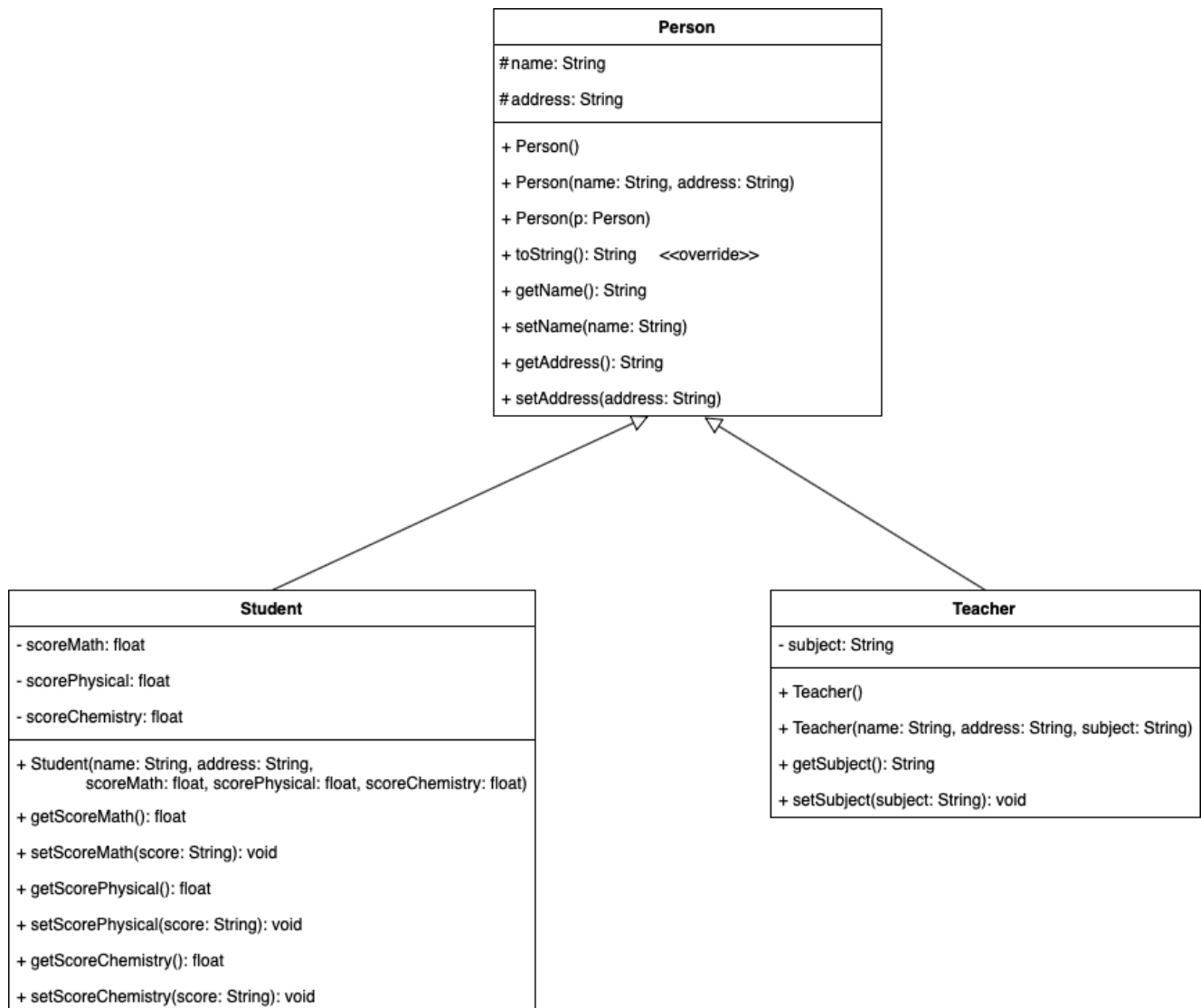
- To refer parent class instance variable.
- To invoke parent class method.
- The **super()** can be used to invoke parent class constructor.

If a class is inheriting the properties of another class. And if the members of the subclass have the names which same as the superclass, to differentiate these variables we use **super** keyword as follows:

- For variable: **super.variableName**
- For method: **super.methodName()**

### III. Sample program

To demonstrate how-to implement inheritance in Java OOP program, we take an example as shown in below diagram:



## Person.java

```
public class Person
{
    protected String name;
    protected String address;

    public Person()
    {
        this.name = "";
        this.address = "";
    }

    public Person(String name, String address)
    {
        this.name = name;
        this.address = address;
    }

    public Person(Person person)
    {
        this.name = person.name;
        this.address = person.address;
    }

    @Override
    public String toString()
    {
        return "Person{" + "name='" + name + "'" + ", address='" + address + "'" + "}";
    }

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

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

    public String getAddress()
    {
        return this.address;
    }

    public void setAddress(String address)
    {
        this.address = address;
    }
}
```

### Teacher.java

```
public class Teacher extends Person
{
    private String subject;

    public Teacher()
    {
        super();
        this.subject = "";
    }

    public Teacher(String name, String address, String subject)
    {
        super(name, address);
        this.subject = subject;
    }

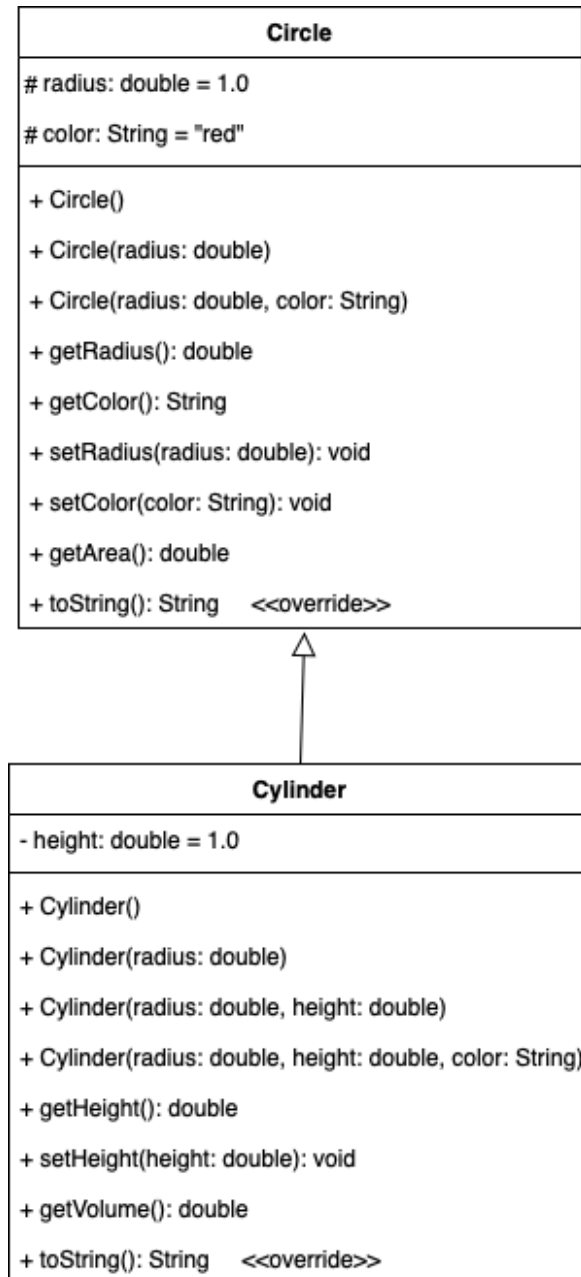
    public String getSubject()
    {
        return this.subject;
    }

    public void setSubject(String subject)
    {
        this.subject = subject;
    }
}
```



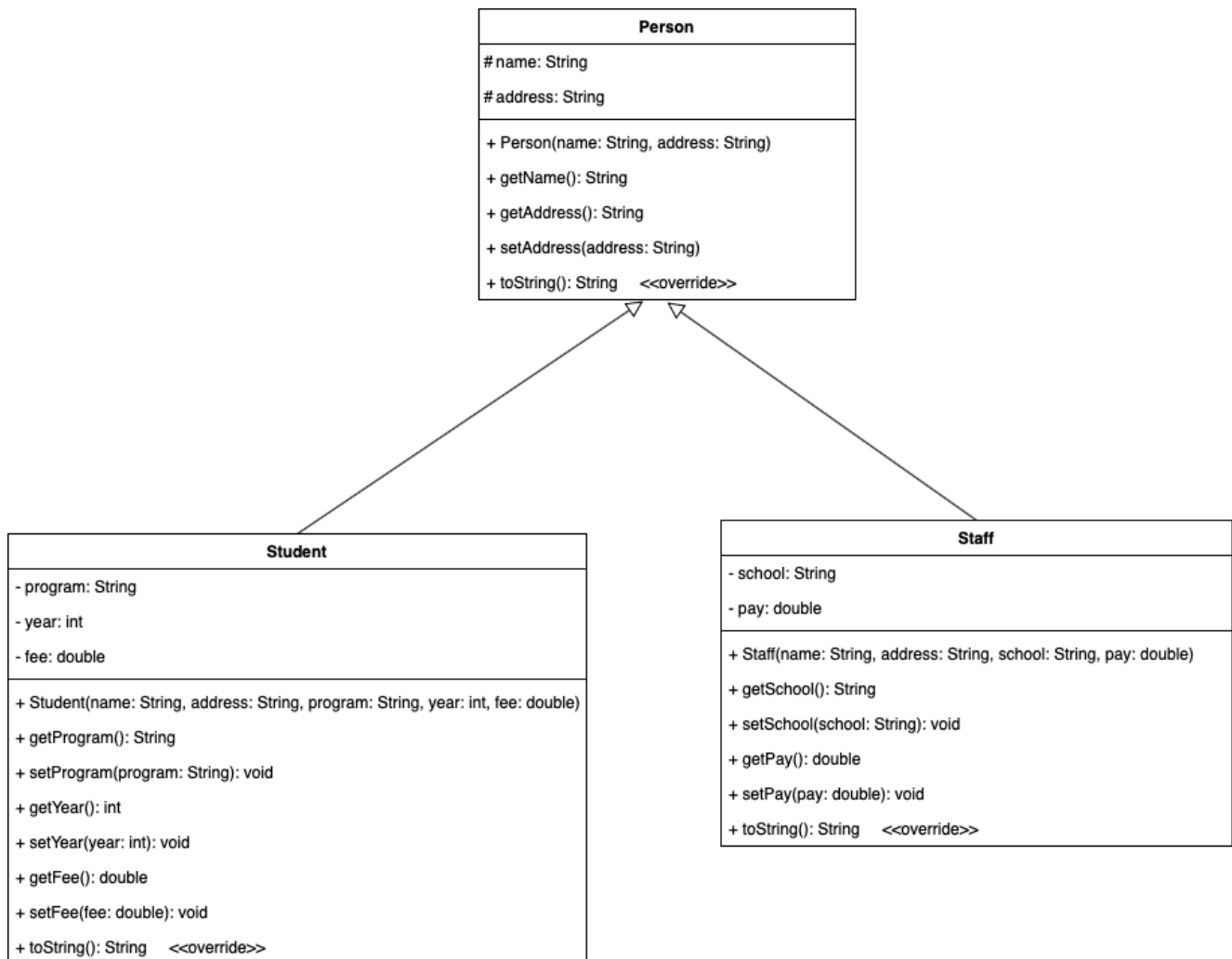
#### IV. Exercise

1. Giving the Circle class and the Cylinder class.



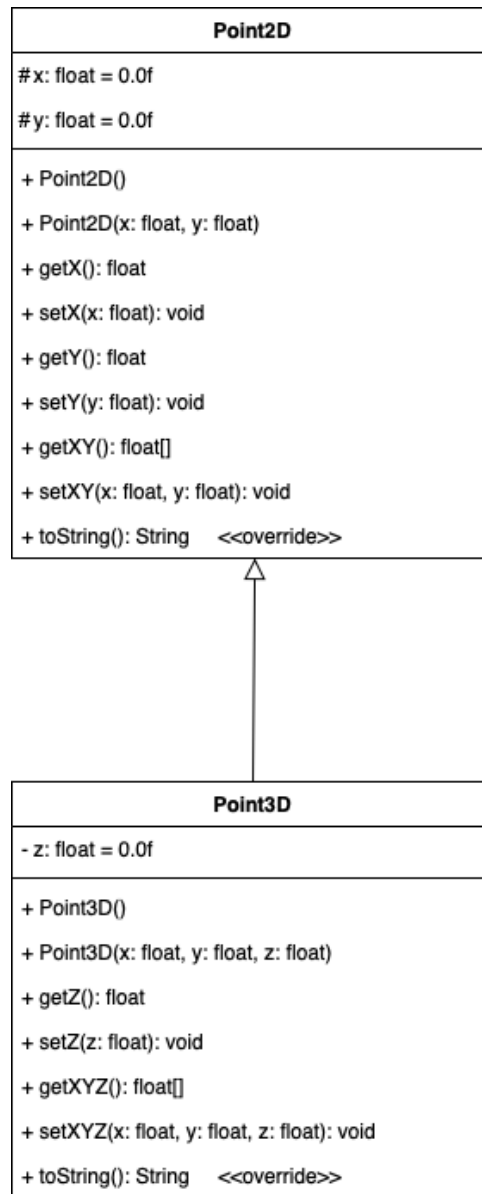
Implement the Java program based on the above diagram.

## 2. Giving superclass Person and its subclasses.



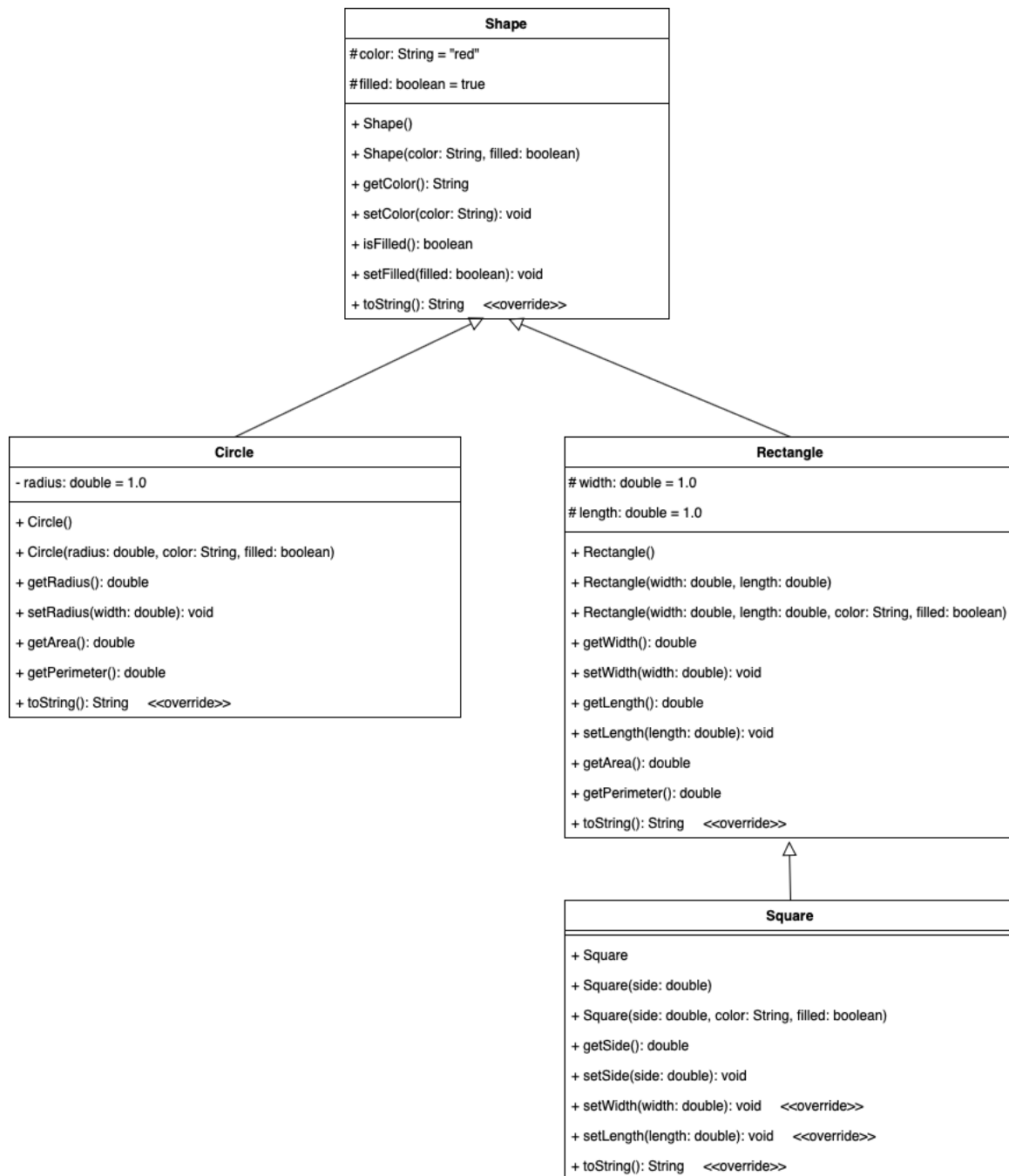
Implement the Java program based on the above diagram.

3. Giving the Point2D class and the Pointed3D class.



Implement the Java program based on the above diagram.

4. Giving superclass Shape and its subclasses.



Implement the Java program based on the above diagram.

5. Implement the Employee class to store the information of employees in the manufacturing company ABC.

**Attributes:**

- **ID**: String
- **fullName**: String
- **yearJoined**: int
- **coefficientsSalary**: double
- **numDaysOff**: int (number of days off in the month)

**Constructors:**

- Constructor with no parameter **Employee()** (ID = 0, fullName = "", yearJoined = 2020, coefficientsSalar = 1.0, numDaysOff = 0)
- Constructor with parameter **Employee(ID: String, fullName: String, coefficientsSalary: double)** (yearJoined = 2020, numDaysOff = 0)
- Constructor with full parameters.

**Methods:**

- **public double getSenioritySalary()**: calculating seniority salary of employees: Know that if an employee works for 5 years or more, the seniority salary is calculated according to the following formula: **seniority salary = years of work \* basic salary / 100**
- **public String considerEmulation()**: write a method to evaluate employee emulation.

If the number of holiday  $\leq 1$  is graded A.

If  $2 \leq$  the number of holidays  $\leq 3$  is graded B.

If the number of holidays  $> 3$  is graded C.

- **public double getSalary()**: write a method for calculating salaries for employees. Know that salary is calculated using the following formula with *basic salary = 1150*:

$$\text{salary} = \text{basic salary} + \text{basic salary} * (\text{salary coefficient} + \text{emulation coefficient}) + \text{seniority salary}$$

- If rated A: emulation coefficient = 1.0
- If rated B: emulation coefficient = 0.75
- If rated C: emulation coefficient = 0.5

6. In addition to the type of employees described in Exercise 5. ABC Company also has a management team to manage all the company's activities called Managers. Let's build a Manager class to let ABC know that managers are also employees of the company. However, due to the role and function, each manager will have a corresponding position, department and salary coefficient by position. The manager is also an employee, we will let Manager class inherit from the Employee class and add some necessary attributes.

**Attributes:**

- The additional attributes include position, department and salary coefficient by position.

**Constructors:**

- Constructor with no parameter **Manager()**: write a default constructor creates a manager like an employee but has the position of head of the office at the administrative office and has a coefficient salary of 5.0.
- Constructor with parameter **Manager(ID: String, fullName: String, coefficientsSalary: double, position: String, salaryCoefficientPosition: double)** (yearJoined = 2020, numDaysOff = 0)
- Constructor with full parameters.

**Methods:**

- **public String considerEmulation()**: override the method to evaluate employee emulation know that employees are always rated A.
- **public double bonusByPosition()**: calculating bonus using the following formula:

$$\text{position bonus} = \text{basic salary} * \text{salary coefficient by position}$$

- **public double getSalary()**: override the method for calculating salaries for employees. Know that the manager's salary is calculated using the following formula:

$$\text{salary} = \text{basic salary} + \text{basic salary} * (\text{salary coefficient} + \text{emulation coefficient}) + \text{seniority salary} + \text{position bonus}$$