

## IS502052: Enterprise Systems Development Concepts

### Lab 1: Java Review

#### I. Introduction

In this first lab, we will review the Java Programming Language, since this course is focus on Java, especially, Java EE<sup>1</sup>, thus you must mastery the fundamental of Java. The topics of this lab include:

- Basic datatypes.
- Loop control.
- Decision making.
- Strings, Arrays.
- Java Object Oriented: *Access modifier, Inheritance, Overriding, Polymorphism, Abstraction, Encapsulation.*

The organization of this lab is structured as follows: **Section 2** presents the Environment Setup, **Section 3** to **Section 7** is for fundamental of Java, **Section 8** is for Java Object Oriented, and **Section 9** is Exercises.

#### II. Environment Setup

To practice all this course's labs, and complete the final project, you need to install the following software, which is available at course's homepage<sup>2</sup>:

- NetBeans 8.2
- jBoss EAP 7.0.0
- PostgreSQL 9.6.3
- pgAdmin 4.1.6
- Java SE Development Kit 8u144<sup>3</sup>

#### III. The Fundamental of Java

##### 1. Basic datatypes

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

---

<sup>1</sup> Java EE: Java Enterprise Edition

<sup>2</sup> <http://it.tdt.edu.vn/~dhphuc/teaching/is502052/>

<sup>3</sup> You should add Java to JAVA\_HOME on Windows OS, visit <https://goo.gl/qNzqgl>

There are two datatypes in Java:

- Primitive Data Types
- Reference/Object Data Types

With primitive data types, there are **8** types supported by Java. Primitive datatypes are predefined by the language and named by a keyword, **Table 1** presents the primitive datatypes.

*Table 1 Java Datatypes*

<i>Type</i>	<i>Description</i>	<i>Default value</i>	<i>Size</i>
<b>boolean</b>	true or false	false	1 bits
<b>byte</b>	twos complement integer	0	8 bits
<b>char</b>	Unicode character	\u0000	16 bits
<b>short</b>	twos complement integer	0	16 bits
<b>int</b>	twos complement integer	0	32 bits
<b>long</b>	twos complement integer	0	64 bits
<b>float</b>	IEEE-754 floating point	0.0	32 bits
<b>double</b>	IEEE-754 floating point	0.0	64 bits

Let's look in the example program below. First, we declare two double variables, *a* and *b*, we then sum them and assign to an integer variable. You should notice that *a* and *b* are casted to integer datatypes before assigned to *sum*.

```
public class Test {
    public static void main(String[] args)
    {
        double a = 3, b = 7;
        int sum = (int) (a + b);

        System.out.println("sum = " + sum);
    }
}
```

*Figure 1 Sample Java program*

## 2. Reference Datatypes

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, *Employee*, *Puppy*, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is *null*.
- A reference variable can be used to refer any object of the declared type or any compatible type.

- For example, `Animal animal = new Animal("giraffe");`.

## IV. Loop Control

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages, as in **Figure 2**.

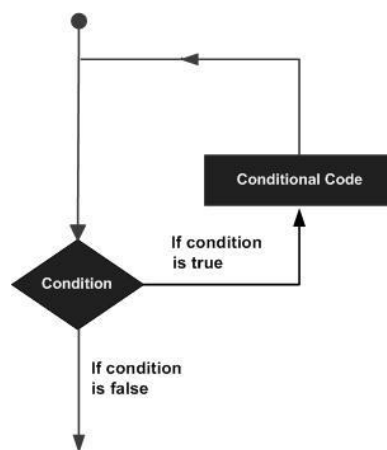


Figure 2 Loop structure

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

- **while** loop
- **for** loop
- **do...while** loop

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

```

public static void main(String[] args)
{
    int x = 10;

    while (x < 100)
    {
        x = x + 10;
    }

    System.out.println("x = " + x);
}
  
```

Figure 3 while 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 is to be repeated.

```
public static void main(String[] args)
{
    int sum = 0;

    for (int i = 0; i < 100; i++)
    {
        sum = sum + i;
    }

    System.out.println("sum = " + sum);
}
```

Figure 4 for loop

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

```
public static void main(String[] args)
{
    int x = 0;

    do {
        x = x + 1;
    } while(x < 10);

    System.out.println("x = " + x);
}
```

Figure 5 do...while loop

**Loop control statements** change execution from its normal sequence. When execution leaves a scope, all objects that were created in that scope are destroyed. Java supports the following control statements:

- **break** statement: terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
- **continue** statement: causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Since Java 5, the **enhanced for loop** has presented, which is mainly used to traverse the collection of elements including arrays.

```
public static void main(String[] args)
{
    int[] x = {2, 3, 5, 7, 11, 13, 17, 19};

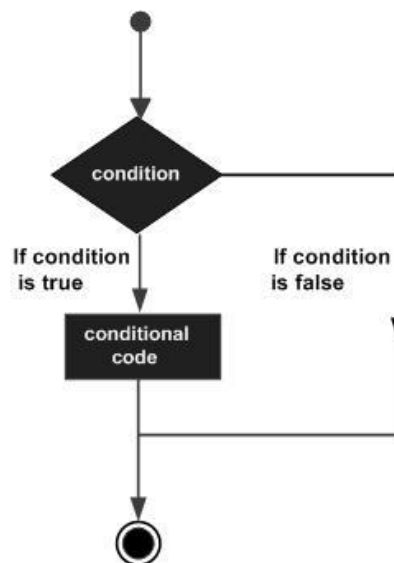
    for(int i : x)
    {
        System.out.println(i);
    }
}
```

Figure 6 Enhanced for loop

## V. Decision Making

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.

Following is the general form of a typical decision-making structure found in most of the programming languages, as in **Figure 7**.



*Figure 7 Decision-making structure*

Java programming language provides following types of decision making statements:

- **if** statement
- **if...else** statement
- **nested if** statement
- **switch** statement

```

public static void main(String[] args)
{
    int x = 10, y = 11;

    if(x % 2 == 0)
    {
        if(y % 2 != 0)
        {
            y = y - 1;
        }
        else
        {
            x = x + y;
        }
    }
    else if(y % 2 == 0)
    {
        x = x - 1;
    }

    System.out.println(x + "\t" + y);
}
  
```

*Figure 8 if...else example*

## VI. String

**String**, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects. The Java platform provides the String class to create and manipulate strings.

<i>Method</i>	<i>Description</i>
<code>char charAt(int index)</code>	returns char value for the particular index
<code>int length()</code>	returns string length
<code>static String format(String format, Object... args)</code>	returns formatted string
<code>static String format(Locale l, String format, Object... args)</code>	returns formatted string with given locale
<code>String substring(int beginIndex)</code>	returns substring for given begin index
<code>String substring(int beginIndex, int endIndex)</code>	returns substring for given begin index and end index
<code>boolean contains(CharSequence s)</code>	returns true or false after matching the sequence of char value
<code>static String join(CharSequence delimiter, CharSequence elements)</code>	returns a joined string
<code>static String join(CharSequence delimiter, Iterable elements)</code>	returns a joined string
<code>boolean equals(Object another)</code>	checks the equality of string with object
<code>boolean isEmpty()</code>	checks if string is empty
<code>String concat(String str)</code>	concatinates specified string
<code>String replace(char old, char new)</code>	replaces all occurrences of specified char value
<code>String replace(CharSequence old, CharSequence new)</code>	replaces all occurrences of specified CharSequence
<code>static String equalsIgnoreCase(String another)</code>	compares another string. It doesn't check case.
<code>String[] split(String regex)</code>	returns splitted string matching regex
<code>String[] split(String regex, int limit)</code>	returns splitted string matching regex and limit

<i>Method</i>	<i>Description</i>
<code>String intern()</code>	returns interned string
<code>int indexOf(int ch)</code>	returns specified char value index
<code>int indexOf(int ch, int fromIndex)</code>	returns specified char value index starting with given index
<code>int indexOf(String substring)</code>	returns specified substring index
<code>int indexOf(String substring, int fromIndex)</code>	returns specified substring index starting with given index
<code>String toLowerCase()</code>	returns string in lowercase.
<code>String toLowerCase(Locale l)</code>	returns string in lowercase using specified locale.
<code>String toUpperCase()</code>	returns string in uppercase.
<code>String toUpperCase(Locale l)</code>	returns string in uppercase using specified locale.
<code>String trim()</code>	removes beginning and ending spaces of this string.
<code>static String valueOf(int value)</code>	converts given type into string. It is overloaded.

## VII. Array

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference, as follows: `dataType[] arrayRefVar;`

Following figure represents an array. Here, array holds ten double values and the indices are from 0 to 9.

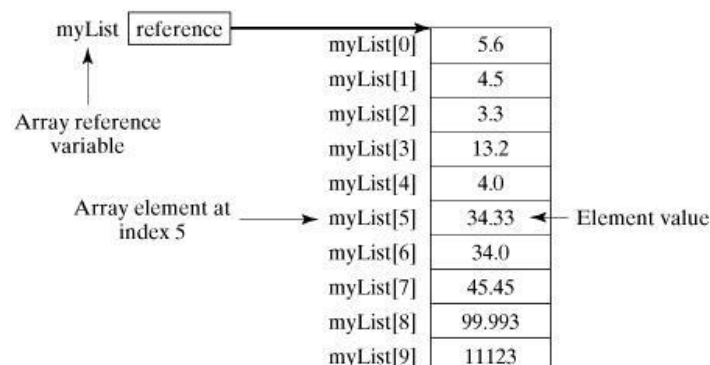


Figure 9 Array

Take a look the following program, illustrating the processing of array.

```
public static void main(String[] args)
{
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++)
    {
        System.out.print(myList[i] + " ");
    }

    // Summing all elements
    double total = 0;
    for (int i = 0; i < myList.length; i++)
    {
        total += myList[i];
    }
    System.out.println("total = " + total);
}
```

Figure 10 Array sample program

## VIII. Java Object Oriented

### 1. Access modifier

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

- Visible to the package, the *default*, no modifiers are needed.
- Visible to the class only (*private*).
- Visible to the world (*public*).
- Visible to the package and all subclasses (*protected*).

### 2. Inheritance

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

#### The *extends* Keyword

**extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

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



## The *super* Keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the *super* keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor and methods from subclass.

The following programs illustrate the inheritance in Java, in **Figure 11** and **Figure 12**.

```
public class Super {  
    protected int x;  
  
    public Super()  
    {  
        this.x = 0;  
    }  
  
    public Super(int x)  
    {  
        this.x = x;  
    }  
  
    public void display()  
    {  
        System.out.println("Super Class, x = " + this.x);  
    }  
}
```

Figure 11 Super class

```
public class Sub extends Super {  
    protected int y;  
  
    public Sub()  
    {  
        super();  
        this.y = 0;  
    }  
  
    public Sub(int x, int y)  
    {  
        super(x);  
        this.y = y;  
    }  
  
    public void display()  
    {  
        super.display();  
        System.out.println("Sub Class, y = " + y);  
    }  
}
```

Figure 12 Sub class

In **Figure 11**, we define a super class and *display* method. In **Figure 12**, we define sub class, including *display* method, an **overriding** version. Now, you should notice that both super class and sub class define *display* method, but in sub class, it calls the *display* method of super class by executing `super.display()`. **Figure 13** is the main function.

```
public class Test {  
  
    public static void main(String[] args)  
    {  
        Super sub = new Sub(10, 100);  
        sub.display();  
    }  
}
```

Figure 13 Main method

## IS-A Relationship

**IS-A** is a way of saying: *This object is a type of that object*. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal {  
}  
  
public class Mammal extends Animal {  
}  
  
public class Reptile extends Animal {  
}  
  
public class Dog extends Mammal {  
}
```

Now, based on the above example, in Object-Oriented terms, the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence: Dog IS-A Animal as well

With the use of the ***extends*** keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass. We can assure that Mammal is actually an Animal with the use of the instance operator.

## HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

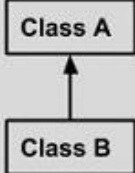
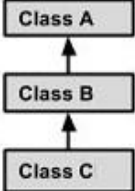
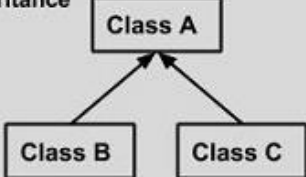
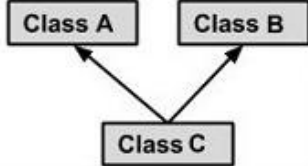
```
public class Vehicle {}  
public class Speed {}  
  
public class Van extends Vehicle {  
    private Speed sp;  
}
```

This shows that class *Van* HAS-A *Speed*. By having a separate class for *Speed*, we do not have to put the entire code that belongs to speed inside the *Van* class, which makes it possible to reuse the *Speed* class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the *Van* class hides the implementation details from the users of the *Speed* class. So, basically what happens is the users would ask the *Van* class to do a certain action and the *Van* class will either do the work by itself or ask another class to perform the action.

## Types of Inheritance

There are various types of inheritance as demonstrated below.

<b>Single Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A]         </pre>	<pre> public class A {     ..... } public class B extends A {     ..... }         </pre>
<b>Multi Level Inheritance</b>  <pre> graph BT     C[Class C] --&gt; B[Class B]     B --&gt; A[Class A]         </pre>	<pre> public class A { ..... } public class B extends A { ..... } public class C extends B { ..... }         </pre>
<b>Hierarchical Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A]     C[Class C] --&gt; A         </pre>	<pre> public class A { ..... } public class B extends A { ..... } public class C extends A { ..... }         </pre>
<b>Multiple Inheritance</b>  <pre> graph BT     C[Class C] --&gt; A[Class A]     C --&gt; B[Class B]         </pre>	<pre> public class A { ..... } public class B { ..... } public class C extends A,B {     ..... } // Java does not support multiple inheritance         </pre>

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal: `public class extends Animal, Mammal{}`.

However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance.

### 3. Overriding

If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

#### Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.

- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared *public* then the overriding method in the sub class cannot be either *private* or *protected*.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared *final* cannot be overridden.
- A method declared *static* cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared *private* or *final*.
- A subclass in a different package can only override the non-final methods declared *public* or *protected*.
- An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

#### **4. Polymorphism**

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class *Object*.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared *final*. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

```
public interface Vehicle {  
    public void run();  
}
```

Figure 14 Interface Vehicle

```
public class Motorbike implements Vehicle {  
    private int wheel;  
  
    public Motorbike()  
    {  
        this.wheel = 0;  
    }  
  
    public Motorbike(int wheel)  
    {  
        this.wheel = wheel;  
    }  
  
    @Override  
    public void run()  
    {  
        System.out.println("Motorbike has " + this.wheel + " wheels. Run!");  
    }  
}
```

Figure 15 Motorbike implements Vehicle

**Figure 14** declares an interface, named by *Vehicle*. This interface contains an abstract method, *run()*, which is not implemented. **Figure 15** declares *Motorbike* class, which implements the *Vehicle* interface. Now, we know how-to implement the *run()* method, since the *Motorbike* is more specific than *Vehicle*.

```
public class Test {  
    public static void main(String[] args)  
    {  
        Vehicle v = new Motorbike(2);  
        v.run();  
    }  
}
```

Figure 16 Main class

```
run:  
Motorbike has 2 wheels. Run!  
BUILD SUCCESSFUL (total time: 0 seconds)  
|
```

Figure 17 Output

## 5. Abstraction

**Abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

### Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain abstract methods, *i.e.*, methods without body, *e.g.*,  
`public abstract void get();`.

- But, if a class has at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you must inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you must provide implementations to all the abstract methods in it.

Look the example below, we define an abstract class, named by *Employee*. Then, we define *Salary* class to extend/implement the *Employee*. You need to notice that there is an abstract method in *Employee*, since we don't know how-to implement it in *Employee*, we make it as abstract method, and then, implement it in *Salary* class.

```
public abstract class Employee {
    private String name;
    private String address;
    private int number;

    public Employee(String name, String address, int number)
    {
        this.name = name;
        this.address = address;
        this.number = number;
    }

    public abstract void mailCheck();

    public String getName()
    { ...3 lines }

    public String getAddress()
    { ...3 lines }

    public int getNumber()
    { ...3 lines }
}
```

Figure 18 An abstract class

```
public class Salary extends Employee {
    private double salary;

    public Salary(String name, String address, int number, double salary)
    {
        super(name, address, number);
        this.salary = salary;
    }

    @Override
    public void mailCheck()
    {
        System.out.println("Mailing a check to " + this.getName()
            + ", at " + this.getAddress()
            + ", the amount is " + this.salary);
    }
}
```

Figure 19 The implementation of abstract class

Now, the main class to test these classes. **Figure 20** will run to error, because *Employee* is an abstract class, you can't instantiate an object.

```
public class Test {
    public static void main(String[] args)
    {
        Employee e = new Employee("John Smith", "TDT", 1);
    }
}
```

Figure 20 An error case of abstraction in Java

To right this wrong, **Figure 21** is a solution.

```
public class Test {
    public static void main(String[] args)
    {
        Employee e = new Salary("John Smith", "TDT", 1, 100);
        e.mailCheck();
    }
}
```

Figure 21 A true case of abstraction in Java

## 6. Encapsulation

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

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.
- Provide **public setter** and **getter** methods to *modify* and *view* the variables values.

## IX. Exercises

1. Write a program to print your name, date of birth, and mobile number.
2. Write a program prompting user to input two integer numbers, then compute and print the results of addition, subtraction, multiplication, division, and remainder.
3. Write a program to compute the perimeter and area of a rectangle with a height provided by user.
4. Write a program to convert specified days into years, weeks and days. (Note: ignore leap year).
5. Write a program to convert the temperature from Celsius to Fahrenheit. (Hint:  $1^{\circ} = 33.8^{\circ}\text{F}$ ).
6. Write a program to return an absolute value of a number.
7. Write a program to check whether a year is a leap year or not.
8. Write a program to find maximum between two numbers.
9. Write a program to find maximum between three numbers.
10. Write a program to check whether a number is even or odd.
11. Write a program to input a character and check whether it is alphanumeric or not.
12. Write a program to input angles of a triangle and check whether triangle is valid or not.
13. Write a program to input marks of five subjects Physics, Chemistry, Biology, Mathematics and Computer. Calculate percentage and grade according to following:
  - Percentage > 90%: Grade A
  - Percentage > 80%: Grade B
  - Percentage > 70%: Grade C
  - Percentage > 60%: Grade D
  - Percentage < 60%: Grade E

14. Write a program to print sum of all even numbers between 1 to  $n$  by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
15. Write a program to print sum of all even numbers between 1 to  $n$  by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
16. Write a program to print table of any number.
17. Write a program to enter any number and calculate sum of all-natural numbers between 1 to  $n$  by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
18. Write a program to find first and last digits of any number.
19. Write a program to calculate sum of digits of any number.
20. Write a program to calculate product of digits of any number.
21. Write a program to count number of digits in any number.
22. Write a program to swap first and last digits of any number.
23. Write a program to enter any number and print its reverse.
24. Write a program to enter any number and check whether the number is palindrome or not.
25. Write a program to check whether a number is Prime number or not. Validating the input, in case the input isn't correct, prompt user to enter it again.
26. Write a program to check whether a number is Armstrong number or not.
27. Write a program to check whether a number is Perfect number or not.
28. Write a program to print all Prime numbers between 1 to  $n$  by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
29. Write a program to print all Armstrong numbers between 1 to  $n$  by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
30. Write a program to print all Perfect numbers between 1 to  $n$  by using three loop structures. Validating the input, in case the input isn't correct, prompt user to enter it again.
31. Write a program to convert Decimal to Binary number system.
32. Write a program to compute the Factorial of  $n$ . Validating the input, in case the input isn't correct, prompt user to enter it again.

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n. \quad (n \geq 0)$$

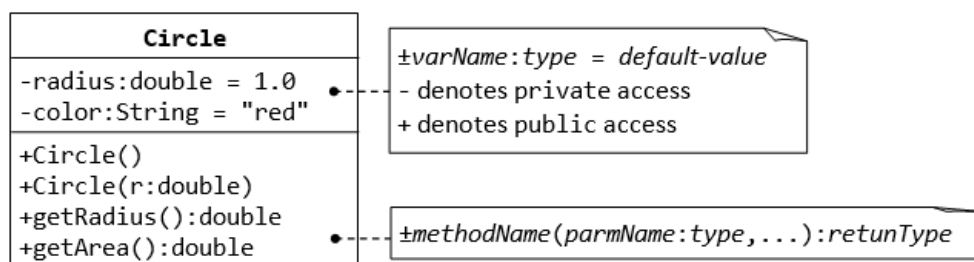
33. Write functions to calculate the following expressions:

a.  $\sum_{i=1}^n \frac{i}{2}$

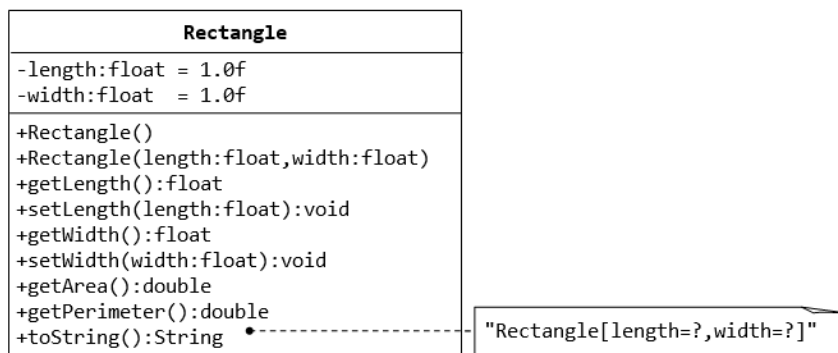
b.  $\sum_{i=1}^n (2i + 1)$



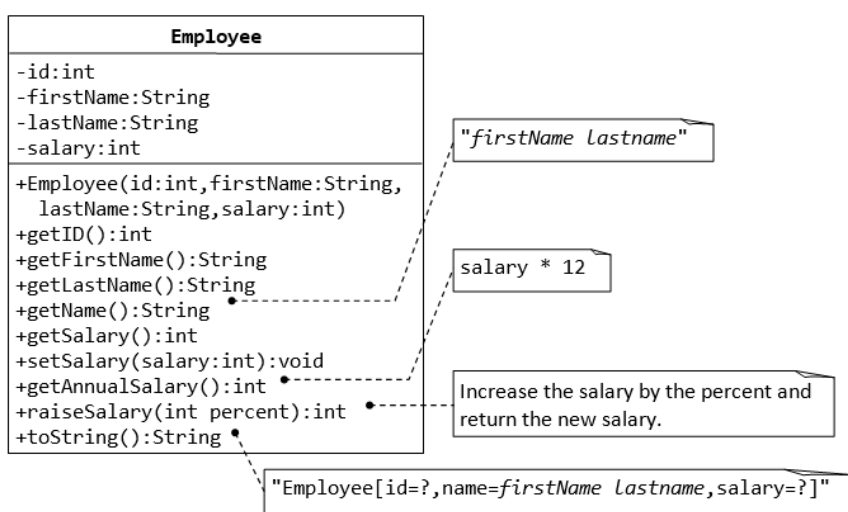
- c.  $\sum_{i=1}^n (i! + 1)$
- d.  $\prod_{i=1}^n i!$
- e.  $\prod_{i=1}^n \frac{2i}{3}$
34. Write function to find the maximum number of an integer array.
  35. Write function to find the minimum number of an integer array.
  36. Write function to sum all numbers of an integer array.
  37. Write function to sum all non-positive numbers of an integer array.
  38. Write function to sum all even numbers of an integer array.
  39. Write function to reverse an array without using any temporary array.
  40. Write program to delete an element from an array at specified position.
  41. Write program to count total number of duplicate elements in an array.
  42. Write program to delete all duplicate elements from an array.
  43. Write program to count frequency of each element in an array.
  44. Write program to merge two arrays to third array.
  45. Write program to put even and odd elements of array into two new separate arrays.
  46. Write program to search an element in an array by providing key value.
  47. Write program to sort array elements in ascending order.
  48. Write program to add two matrices.
  49. 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.



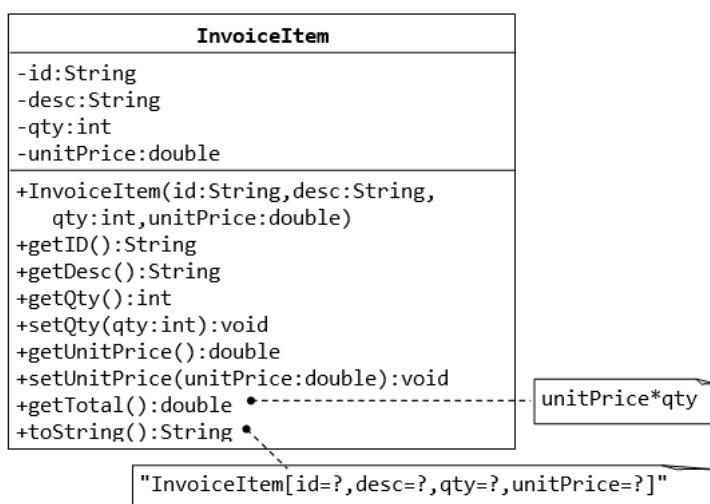
**50.** The **Rectangle** Class is defined as below figure. Program the class follows by the UML diagram.



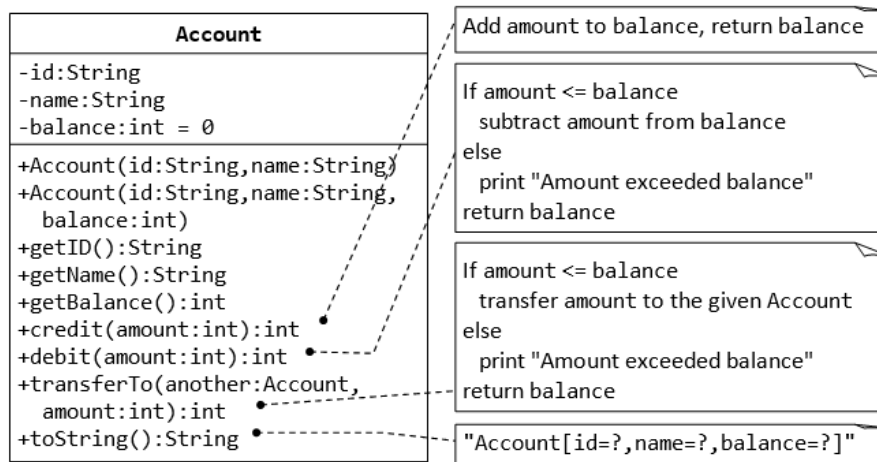
**51.** The **Employee** Class is defined as below figure. Program the class follows by the UML diagram.



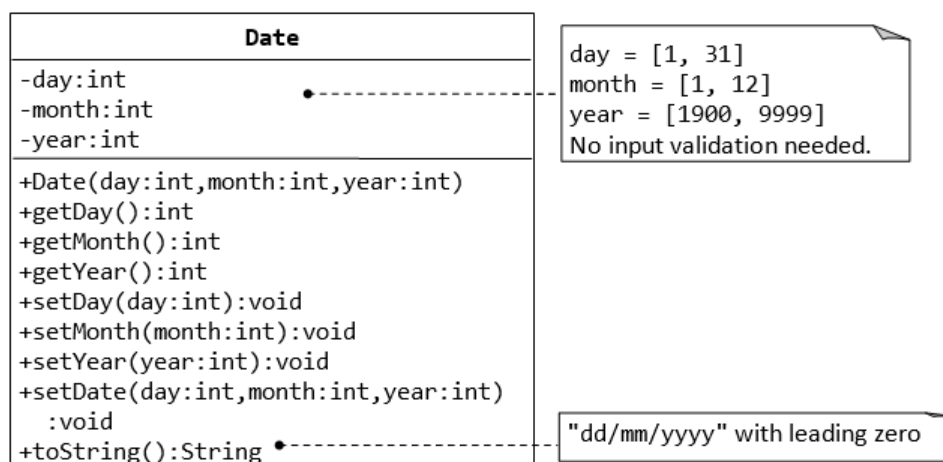
**52.** The **InvoiceItem** Class is defined as below figure. Program the class follows by the UML diagram.



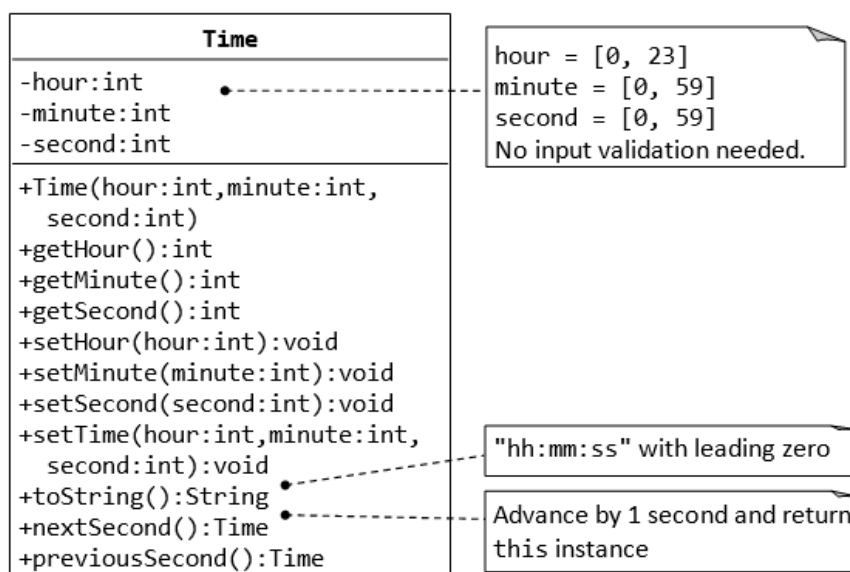
53. The **Account** Class is defined as below figure. Program the class follows by the UML diagram.



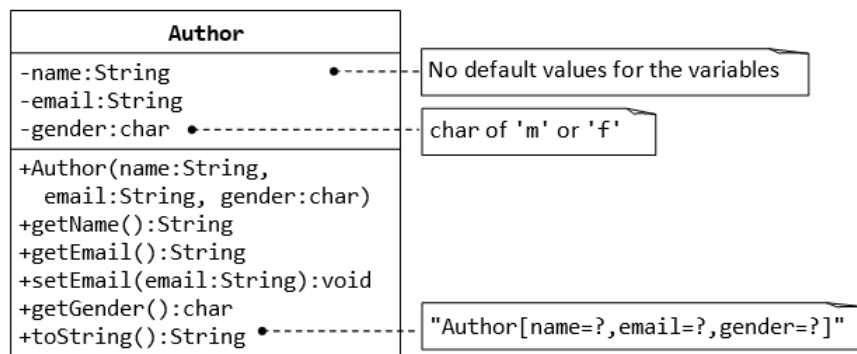
54. The **Date** Class is defined as below figure. Program the class follows by the UML diagram.



55. The **Time** Class is defined as below figure. Program the class follows by the UML diagram.

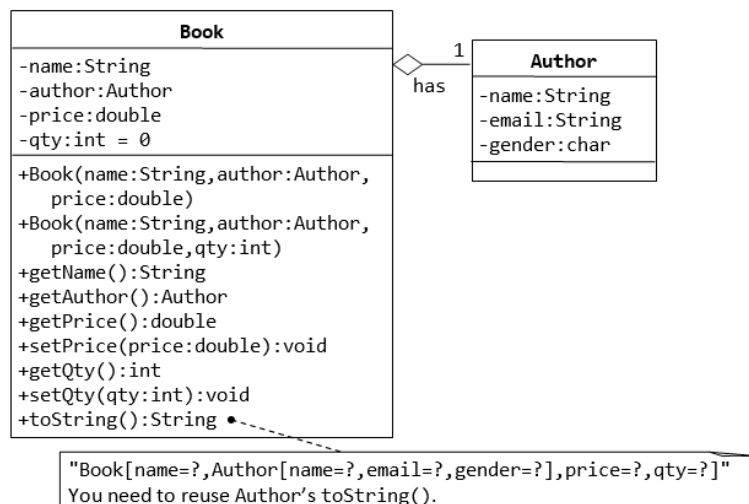


## 56. The **Author** and **Book** Classes:



A class called **Author** (as shown in the class diagram) is designed to model a book's author. It contains:

- Three private instance variables: *name* (String), *email* (String), and *gender* (char of either 'm' or 'f');
- One constructor to initialize the *name*, *email* and *gender* with the given values;
- public getters/setters: *getName()*, *getEmail()*, *setEmail()*, and *getGender()*;
- A *toString()* method that returns "*Author[name=?,email=?,gender=?]*" format.



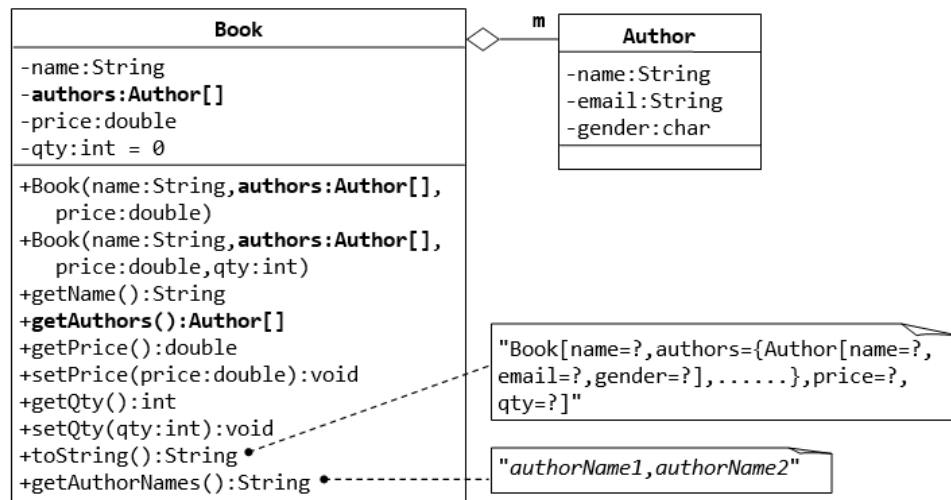
A class called **Book** is designed (as shown in the class diagram) to model a book written by one author. It contains:

- Four private instance variables: *name* (String), *author* (of the class Author you have just created, assume that a book has one and only one author), *price* (double), and *qty* (int);
- Two constructors:
  - + public Book (String name, Author author, double price) {}
  - + public Book (String name, Author author, double price, int qty) {}
- Public methods *getName()*, *getAuthor()*, *getPrice()*, *setPrice()*, *getQty()*, *setQty()*.

- A `toString()` that returns `"Book[name=?,Author[name=?,email=?,gender=?],price=?,qty=?"]`. You should reuse **Author**'s `toString()`.

Require: write the **Author** class and the **Book** class (which uses the **Author** class written earlier).

### 57. **Book** and **Author** Classes - An Array of Objects as an Instance Variable



In the earlier exercise, a book is written by one and only one author. In reality, a book can be written by one or more author. Modify the **Book** class to support one or more authors by changing the instance variable `authors` to an **Author** array.

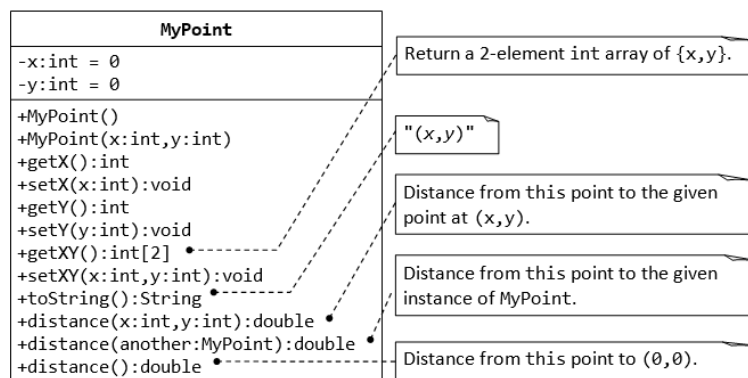
Notes:

- The constructors take an array of **Author** (i.e., `Author[]`), instead of an **Author** instance. In this design, once a **Book** instance is constructor, you cannot add or remove author.
- The `toString()` method shall return `"Book[name=?,authors={Author[name=?,email=?,gender=?],.....},price=?,qty=?]"`.

You are required to:

- Write the code for the **Book** class. You shall re-use the **Author** class written earlier.
- Write a test driver (called **TestBook**) to test the **Book** class.

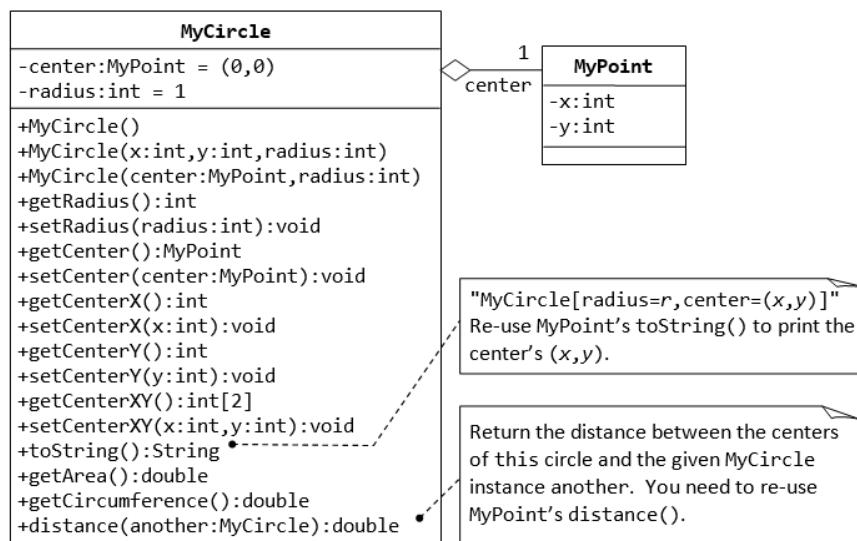
### 58. The **MyPoint** Class is defined as below figure. Program the class follows by the UML diagram.



A class called **MyPoint**, which models a 2D point with  $x$  and  $y$  coordinates, is designed as shown in the class diagram. It contains:

- Two instance variables  $x$  (int) and  $y$  (int).
- A default (or "no-argument") constructor that construct a point at the default location of (0, 0).
- A overloaded constructor that constructs a point with the given  $x$  and  $y$  coordinates.
- *Getter* and *setter* for the instance variables  $x$  and  $y$ .
- A method `setXY()` to set both  $x$  and  $y$ .
- A method `getXY()` which returns the  $x$  and  $y$  in a 2-element int array.
- A `toString()` method that returns a string description of the instance in the format " $(x, y)$ ".
- A method called `distance(int x, int y)` that returns the distance from this point to another point at the given  $(x, y)$  coordinates.
- An overloaded `distance(MyPoint another)` that returns the distance from this point to the given **MyPoint** instance (called another).
- Another overloaded `distance()` method that returns the distance from this point to the origin (0,0).

#### 59. The **MyCircle** and **MyPoint** Classes (cont. from Ex. 58)



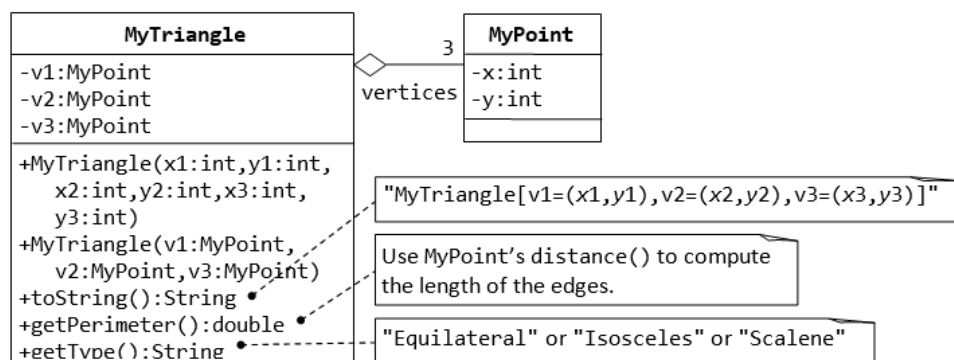
A class called **MyCircle**, which models a circle with a center  $(x,y)$  and a radius, is designed as shown in the class diagram. The **MyCircle** class uses an instance of **MyPoint** class (created in the previous exercise) as its center.

The class contains:

- Two private instance variables: *center* (an instance of **MyPoint**) and *radius* (int).
- A constructor that constructs a circle with the given center's  $(x, y)$  and radius.
- An overloaded constructor that constructs a **MyCircle** given a **MyPoint** instance as *center*, and *radius*.
- A default constructor that construct a circle with center at  $(0,0)$  and radius of 1.
- Various *getters* and *setters*.
- A *toString()* method that returns a string description of this instance in the format `"MyCircle[radius=r,center=(x,y)]"`. You shall reuse the *toString()* of **MyPoint**.
- *getArea()* and *getCircumference()* methods that return the area and circumference of this circle in double.
- A *distance(MyCircle another)* method that returns the distance of the centers from this instance and the given **MyCircle** instance. You should use **MyPoint**'s *distance()* method to compute this distance.

Write the **MyCircle** class. Also write a test driver (called **TestMyCircle**) to test all the public methods defined in the class.

## 60. The **MyTriangle** and **MyPoint** Classes (cont. from Ex. 58)



A class called **MyTriangle**, which models a triangle with 3 vertices, is designed as shown. The **MyTriangle** class uses three **MyPoint** instances (created in the earlier exercise) as its three vertices.

It contains:

- Three private instance variables `v1`, `v2`, `v3` (instances of **MyPoint**), for the three vertices.
- A constructor that constructs a **MyTriangle** with three set of coordinates, `v1=(x1, y1)`, `v2=(x2, y2)`, `v3=(x3, y3)`.
- An overloaded constructor that constructs a **MyTriangle** given three instances of **MyPoint**.
- A `toString()` method that returns a string description of the instance in the format `"MyTriangle[v1=(x1,y1),v2=(x2,y2),v3=(x3,y3)]"`.
- A `getPerimeter()` method that returns the length of the perimeter in double. You should use the `distance()` method of **MyPoint** to compute the perimeter.
- A method `printType()`, which prints `"equilateral"` if all the three sides are equal, `"isosceles"` if any two of the three sides are equal, or `"scalene"` if the three sides are different.

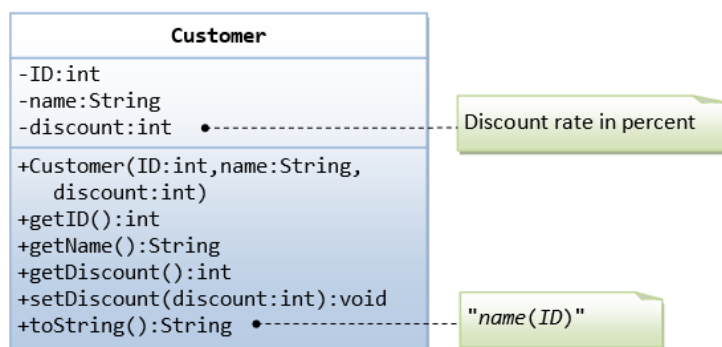
Write the **MyTriangle** class. Also write a test driver (called **TestMyTriangle**) to test all the public methods defined in the class.

## 61. The **MyRectangle** and **MyPoint** Classes (cont. from Ex. 58)

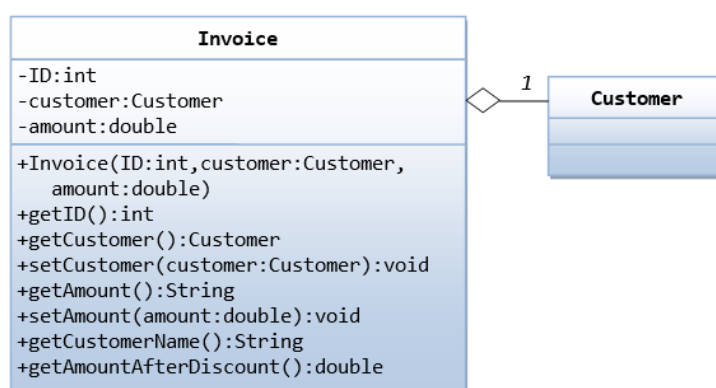
Require: design a **MyRectangle** class which is composed of two **MyPoint** instances as its *top-left* and *bottom-right* corners. Draw the class diagrams, write the codes, and write the test drivers.

## 62. The **Customer** and **Invoice** classes



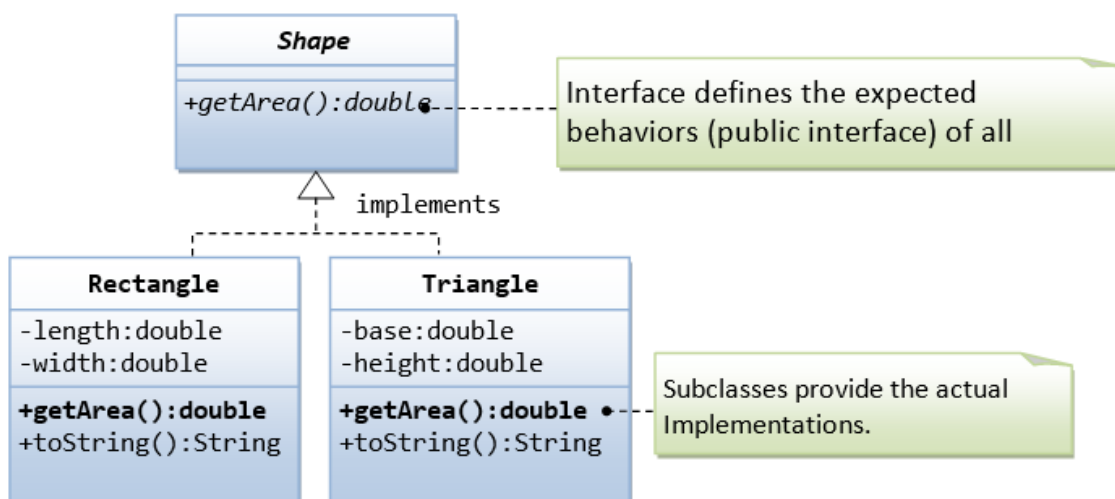


The **Customer** class models a customer is design as shown in the class diagram. Write the codes for the **Customer** class and a test driver to test all the public methods.



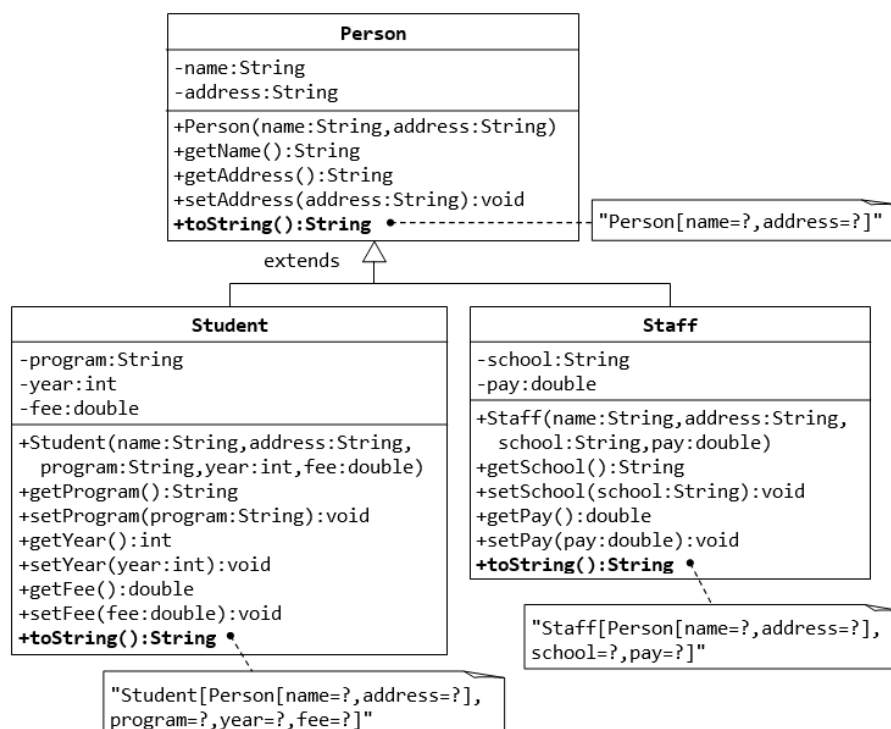
The **Invoice** class, design as shown in the class diagram, composes a **Customer** instance (written earlier) as its member. Write the codes for the **Invoice** class and a test driver to test all the public methods.

**63.** The **Shape** interface; **Rectangle** and **Triangle** classes:



Implement the Java program to illustrate this relationship.

**64.** The **Person** interface; **Student** and **Staff** classes:



Create a main method and instantiate a list of **Person** which each member can be **Student** or **Staff**.

## 65. The abstract class **Shape**; **Circle**, **Rectangle** and **Square** classes:



**66.** The **GeometricObject** and **Resizable** interfaces; the **Circle** and **ResizableCircle** classes:

