TP 2: Object Oriented Programming

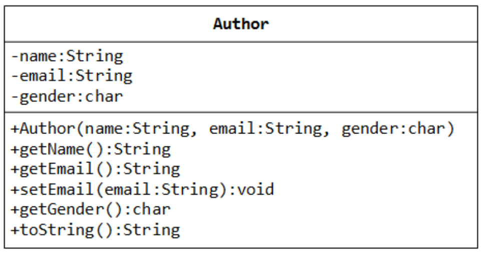
Problem 1: Printing an Array using a Method

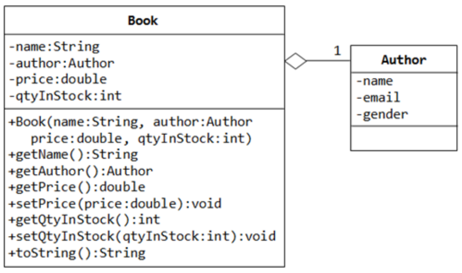
Write a program and use overloaded methods for printing different types of array (integer, double and character). Create two classes:

- ***Launcher***: contains the ***main*** method.
- ***CustomArrayPrinter****: contains your overloaded methods*

Instantiate and test your ***CustomArrayPrinter*** inside the ***main*** method to print different type of array.

Problem 2: Author and Book





Write ***Author*** class designed as following:

- 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;
- Getters and setters: getName(), getEmail() and getGender(). There are no setters for name and gender because these attributes cannot be changed.

Also write a test program to test the Author constructor and call the public methods of the Author class. (including toString() method)

```
Author anAuthor = new Author("Tan Ah Teck", "ahteck@somewhere.com", 'm');
```

Write **Book** class designed as following:

- Four private instance variables: name (String), author (of the class Author you have just created), price (double), and qtyInStock (int). Assuming that each book is written by one author.
- One constructor which constructs an instance with the values given.
- Getters and setters: getName(), getAuthor(), getPrice(), setPrice(), getQtyInStock(), setQtyInStock(). Again there is no setter for name and author.

Also write a test program to test the constructor and public methods in the class Book. Take Note that you have to construct an instance of Author before you can construct an instance of Book. E.g.,

```
Author anAuthor = new Author(......);
Book aBook = new Book("Java for dummy", anAuthor, 19.95, 1000);
// Use an anonymous instance of Author
Book anotherBook = new Book("more Java for dummy", new Author(......), 29.95, 888);
```

1. Print the book name, price and qtyInStock from a Book instance.
2. After obtaining the "Author" object, print the Author (name, email & gender) of the book.

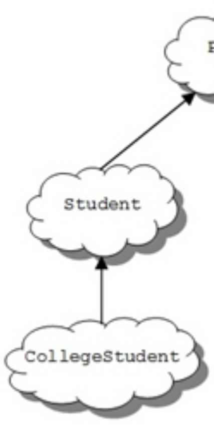
Problem 3: Inheritance

A HighSchool application has two classes: The **Person** superclass and the **Student** subclass. Using inheritance, in this lab you will create two new classes, **Teacher** and **CollegeStudent**.

A **Teacher** will be like Person but will have additional properties such as *salary* (the amount the teacher earns) and *subject* (e.g. “Computer Science”, “Chemistry”, “English”, “Other”).

The **CollegeStudent** class will extend the Student class by adding a *year* (current level in college) and *major* (e.g. “Electrical Engineering”, “Communications”, “Undeclared”).

The inheritance hierarchy would appear as follows:



Listed below is the Person base class from the lesson to be used as a starting point for the Teacher class:

```
class Person {
protected String name;   // name of the person
protected int age;       // person’s age
protected String gender;  // “M” for male, “F” for female
public Person(String name, int age, String gender)  {
                          //affect each parameter to the corresponding attribute
             }
public String toString()  {
return name + “, age: ” + age + “, gender: ” + gender;
}
}
```

The Student class is derived from the Person class and used as a starting point for the CollegeStudent class:

```
class Student extends Person {
protected String studentId;    // Student Id Number
protected double gradePointAverage;      // grade point average
public Student(String name, int age, String gender, String studentId, double gradePointAverage)  {
// use the super class’ constructor
// initialize what’s new to Student
}
}
```

Assignment:

1. Add methods to "set" and "get" the instance variables in the Person class.

2. Add methods to "set" and "get" the instance variables in the Student class.

3. Write a Teacher class that extends the parent class Person.

 a. Add instance variables to the class for *subject* (e.g. "Computer Science", "Chemistry", "English", "Other") and *salary* (the teacher's annual salary). *Subject* should be of type String and *salary* of type double. Choose appropriate names for the instance variables.

 b. Write a constructor for the Teacher class. The constructor will use five parameters to initialize name, age, gender, *subject*, and *salary*. Use the super reference to use the constructor in the Person superclass to initialize the inherited values.

 c. Write "setter" and "getter" methods for all of the class variables. For the Teacher class they would be: getSubject, getSalary, setSubject, and setSalary.

 d. Write the toString() method for the Teacher class. Use a super reference to do the things already done by the superclass.

4. Write a CollegeStudent subclass that extends the Student class.

 a. Add instance variables to the class for *major* (e.g. "Electrical Engineering", "Communications", "Undeclared") and *year* (e.g. FROSH = 1, SOPH = 2, …). *Major* should be of type String and *year* of type int. Choose appropriate names for the instance variables.

 b. Write a constructor for the CollegeStudent class. The constructor will use seven parameters to initialize name, age, gender, studentId, gradePointAverage, *year*, and *major*. Use the super reference to use the constructor in the Student superclass to initialize the inherited values.

 c. Write "setter" and "getter" methods for all of the class variables. For the CollegeStudent class they would be: getYear, getMajor, setYear, and setMajor.

 d. Write the toString() method for the CollegeStudent class. Use a super reference to do the things already done by the superclass.

5. Write a testing class with a main() that constructs all of the classes (Person, Student, Teacher, and CollegeStudent) and calls their toString() method.

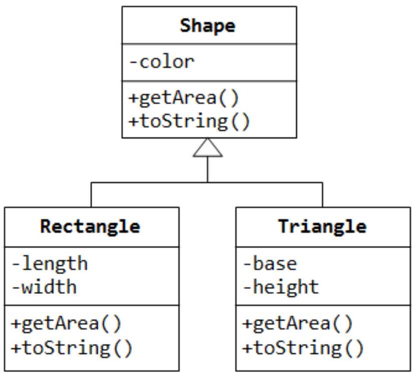
Sample usage would be:

```
Person bob = new Person("Coach Bob", 27, "M");
System.out.println(bob);

Student lynne = new Student("Lynne Brooke", 16, "F", "HS95129", 3.5);
System.out.println(lynne);

Teacher mrJava = new Teacher("Duke Java", 34, "M", "Computer Science", 50000);|
System.out.println(mrJava);

CollegeStudent ima = new CollegeStudent("Ima Frosh", 18, "F", "UCB123", 4.0, 1, "English");

System.out.println(ima);
```

Problem 4: Polymorphism Example

Polymorphism is very powerful in OOP to *separate the interface and implementation* to allow the programmer to *program complex system*.

Consider the following example,



Our program uses many kinds of shapes, such as triangle, rectangle and so on. You should design a super class called Shape, which defines the public interface (or behaviors) of all the shapes as mentioned in the above class diagram.
We would like all the shapes to have a method called getArea(), which returns the area of that particular shape.

Define a **Shape** class as mentioned in the above class diagram.

```
public abstract class Shape {
  // Instance variable color
  // create a constructor for Shape with given color
  // write toString() to print the  Shape of color
  // All shapes must has a method called getArea(), write an abstract method getArea.
}
```

Then you can then derive subclasses, such as **Triangle** and **Rectangle**, from the super class Shape. (Refer the above class diagram)

```
public class Rectangle extends Shape {
// Instance variables
 // Create a constructor with given color, length & width
// call super class constructor Shape(color)
// Override toString() to return the  length, width of the Rectangle object and also call super.toString()
 to print the color of the Rectangle.
// Override getArea() and provide implementations for calculating the area of the Rectangle…
}
```

```
public class Triangle extends Shape {
 // Instance variables
// Create a constructor with given color, base & height
// call super class constructor Shape(color)
// Override toString() to return the  base, height of the Triangle object and also call super.toString()
 to print the color of the Triangle.
 // Override getArea() and provide implementations for calculating the area of the Triangle… use the formula
(0.5*base*height)
}
```

The subclasses override the getArea() method inherited from the super class, and provide the proper implementations for getArea().

Finally, create a **Launcher** class in our application, then create references of Shape, and assign them instances of subclasses. And call the *getArea()* methods of Rectangle & Triangle by invoking Shape references.

```
public class Launcher  {
  public static void main(String[] args) {

     //Shape s1, s2;
     // System.out.println(s1); calls toString()…
     //   System.out.println("Area is " + s1.getArea());
     // System.out.println(s2); calls toString()…
     // System.out.println("Area is " + s2.getArea());
  }
}
```

Note:
The beauty of this code is that *all the references are from the super class* (i.e., *programming at the interface level*). You could instantiate different subclass instance, and the code still works. You could extend your program easily by adding in more subclasses, such as Circle, Square, etc, with ease.

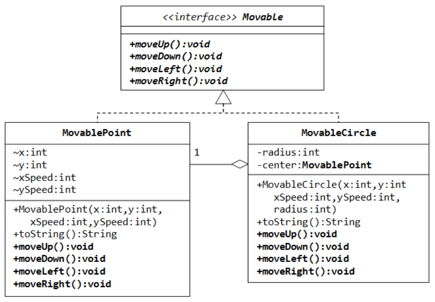
Nonetheless, the above definition of Shape class poses a problem, if someone instantiate a Shape object and invoke the getArea() from the Shape object, the program breaks.

```
public class Launcher {
  public static void main(String[] args) {
    // Constructing a Shape instance poses problem!
    Shape s3 = new Shape("green");
    System.out.println(s3);
    System.out.println("Area is " + s3.getArea());
  }
}
```

This is because the Shape class is meant to provide a common interface to all its subclasses, which are supposed to provide the actual implementation. We do not want anyone to instantiate a Shape instance. This problem can be resolved by using the so-called abstract ***class****.*

Problem 5: Interfaces

Suppose that we have a set of objects with some common behaviors: they could move up, down, left or right. The exact behaviors (such as how to move and how far to move) depend on the objects themselves.



One common way to model these common behaviors is to define an interface called **Movable**, with abstract methods

moveUp(), moveDown(), moveLeft(), moveRight()

The classes that implement the **Movable** interface will provide actual implementation to these abstract methods.

Let's write two concrete classes - MovablePoint and MovableCircle - that implement the Movable interface.

The code for the interface Movable is straight forward.

```
public interface Movable {  // saved as "Movable.java"
   public void moveUp();
   ......
}
```

For the MovablePoint class, declare the instance variable x, y, xSpeed and ySpeed with package access as shown with '~' in the class diagram (i.e., classes in the same package can access these variables directly). For the MovableCircle class, use a MovablePoint to represent its center (which contains four variable x, y, xSpeed and ySpeed).

In other words, the MovableCircle composes a MovablePoint, and its radius.

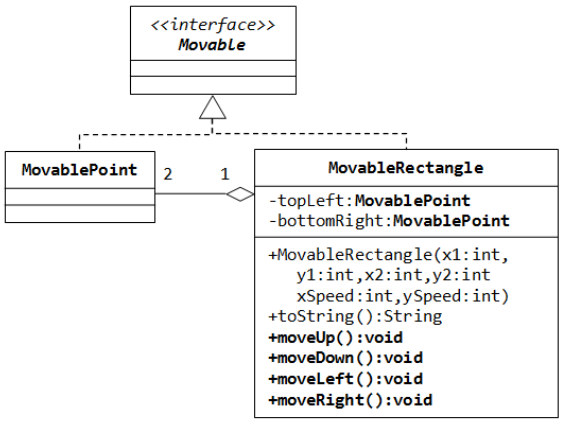
```
public class MovablePoint implements Movable { // saved as "MovablePoint.java"
   // instance variables
   int x, y, xSpeed, ySpeed;     // package access

   // Constructor
   public MovablePoint(int x, int y, int xSpeed, int ySpeed) {
      this.x = x;
      ......
   }
   ......

   // Implement abstract methods declared in the interface Movable
   @Override
   public void moveUp() {
      y -= ySpeed;   // y-axis pointing down for 2D graphics
   }
   ......
}
```

```
public class MovableCircle implements Movable { // saved as "MovableCircle.java"
  // instance variables
  private MovablePoint center;   // can use center.x, center.y directly
                                  //  because they are package accessible
  private int radius;

  // Constructor
  public MovableCircle(int x, int y, int xSpeed, int ySpeed, int radius) {
    // Call the MovablePoint's constructor to allocate the center instance.
    center = new MovablePoint(x, y, xSpeed, ySpeed);
    ......
  }
  ......

  // Implement abstract methods declared in the interface Movable
  @Override
  public void moveUp() {
    center.y -= center.ySpeed;
  }
  ......
}
```

Write a test program and try out these statements:

```
Movable m1 = new MovablePoint(5, 6, 10, 4);     // upcast
System.out.println(m1);
m1.moveLeft();
System.out.println(m1);

Movable m2 = new MovableCircle(2, 1, 2, 4, 20); // upcast
System.out.println(m2);
m2.moveRight();
System.out.println(m2);
```

Write a new class called MovableRectangle, which composes two MovablePoints (representing the top-left and bottom-right corners) and implementing the Movable Interface. Make sure that the two points has the same speed.



What is the difference between an interface and an abstract class?