Lab 4
Inheritance

Object-oriented programming allows code re-use by allowing a class to adopt the features of another class using **inheritance**.

The class that inherits from another class is called a **sub-class** or a **child class**. The class from which another class is inherited from is called its **super class** or **parent class**. For example, the class *Person* is the parent class of the *Student* and *Faculty* classes if *Student* and *Faculty* are subclasses of *Person*.

Inheritance creates a relationship between classes in which subclass objects can be used as if they were objects of its super class. As such, a student is a person and a faculty is also a person; thus sub-classing creates what is called an **"is-a" relationship**.

Sub-classing in Java happens by extending a class (using the keyword **extends**). A class is a data structure that implements methods and fields. Java implements **single-class** inheritance (that is, a class can only "extend" one other class).

In this lab, we will have exercises covering sub-classing.

1. Shapes

In this exercise, we will write classes for 3 different shapes: circles, triangles and quadrilaterals (which are polygons with 4 sides).

Shapes have things in common. For example, they have a name, all have a series of points for vertices (triangles have 3, quadrilaterals have 4, and circles have 1 for their center), and all have a perimeter. A common super-class should store these characteristics. As shown in Figure 1, this parent class is the class "Shape".

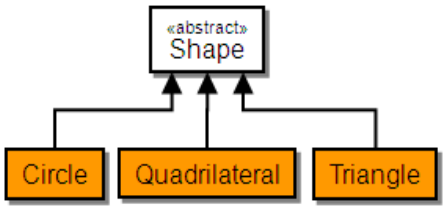

Figure 1. Class hierarchy of shapes.

Even though it is useful to have a "Shape" class, it is hard to imagine what a "Shape" object could be. We can imagine triangles and squares; but somehow a shape object is hard to describe. This is a strong indication that our "Shape" class should be abstract: if a class acts as a common repository of

features but does not make sense having its own objects, then it should be "abstract". As such, we will make "Shape" an "abstract" class.

Exercise 1

In this exercise you will finish implementing the **Shape** abstract class, and write the classes Circle, Triangle and Quadrilateral.

1. Create a **Lab04PartA** project.
2. Download **Shape.java** from the class website, and import it into the project.
3. Open the Shape class in the editor. This class defines 2 private fields: **name** (a string representing the name of the shape) and **points** (an array of **Point** values to store the vertices of the shape –the **Point** class is already defined in Java as **java.awt.Point** – check the Java API to see its features). The Shape class is not complete. Implement all methods that have a "**TODO**" comment (you should not change their definitions). The purpose of these methods is defined below.
   - The **constructor** receives a String to initialize the name of the shape. This constructor is **protected** and thus only visible to subclasses.
     ```
     protected Shape(String aName)
     ```
   - The method **getName** returns the name of the shape. This method is **final** and thus cannot be overridden in subclasses.
     ```
     public final String getName()
     ```
   - The method **setPoints** receives an array of points and assigns it to the field **points**. This method is **protected** (only visible to subclasses) and **final** (cannot be overridden in subclasses).
     ```
     protected final void setPoints(Point[] points)
     ```
   - The method **getPoints** returns the vertices of the shape (given by the array "points"). This method is **final** (cannot be overridden in subclasses).
     ```
     public final Point[] getPoints()
     ```
   - The method **getPerimeter** is **abstract** and should not be implemented here. Rather it will be overridden in subclasses.
   - The static method **getDistance** receives 2 points and calculates the distance between them. To this end, we can use a theorem that states that the distance between 2 points is the square root of the sum of the squares of the absolute differences between their x and y locations, i.e., $\text{sqrt}(\Delta x^2 + \Delta y^2)$. For example, given the points A at position (2, 1) and B at position (5, 6) (shown in Figure 2), we first calculate the absolute difference between their x and y positions, which are $\Delta x = \text{abs}(2 - 5) = 3$, and $\Delta y = \text{abs}(1 - 6) = 5$; we get their squares, which are $\Delta x^2 = 3^2 = 9$, and $\Delta y^2 = 5^2 = 25$; and lastly, we calculate the square root of their sum, i.e., $\text{sqrt}(\Delta x^2 + \Delta y^2) = \text{sqrt}(9 + 25) = \text{sqrt}(34) = 5.83$. As such, the distance between points A and B in this example is 5.83.

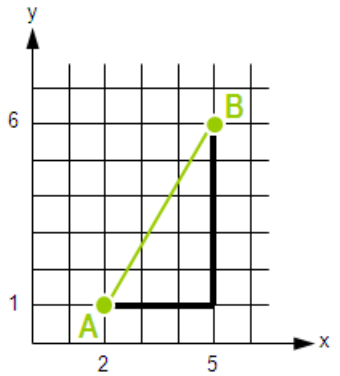

Figure 2. Distance between 2 points using the Pythagorean Theorem.

Hint: the **Math class** defines methods to calculate square root, power and absolute values. Look up **java.awt.Point** in the Java API to know how to get the x and y values of a Point.

- Be reminded that you should not modify the definition of any methods or fields.

5. Create a class **Circle** that subclasses from Shape. This class should define a private double field to store the radius of the circle. In addition, it should implement the following methods:
   - A public **constructor** receiving a point object (representing the center of the circle) and a double (for the radius). Use the inherited *setPoints* to store it. If a negative radius is passed as a parameter, the radius should get a value of 0. The name of circles should always be "Circle".
   - The method **getRadius**, which should return the radius.
   - The method **getPerimeter**, which calculates and returns the perimeter of the circle as "2 * π * radius". The constant π can be found in the Java Math class.
   - No other methods or fields should be implemented.
6. Create a class **Quadrilateral** that subclasses from Shape. It should implement the methods below.
   - A public **constructor** receiving an array of points (representing the vertices of the quadrilateral). This array can have 4 points or more. But since a quadrilateral needs only 4 points you will create a new array of 4 points and copy the first 4 points from the given array. Use the inherited *setPoints* to store the array you created. The name of quadrilaterals should be "Quadrilateral".
   - The method **getPerimeter**, which calculates and returns the perimeter as the sum of the length of all 4 sides (Hint: use the static method *getDistance* in Shape to calculate these lengths).
   - No other methods or fields should be implemented.
7. Create a class **Triangle** that subclasses from Shape. It should implement the methods below:
   - A public **constructor** receiving an array of points (representing the vertices of the triangle). Assume the array has at least 3 points. Create a new array of 3 points, copy the first 3 points from the given array into your new array, and use the inherited *setPoints* to store this new array. The name of triangles should always be "Triangle".
   - The method **getPerimeter**, which calculates and returns the perimeter as the sum of the length of all 3 sides (Hint: use the static method *getDistance* in Shape to calculate these lengths).
   - No other methods or fields should be implemented.
8. Download the JUnit files **ShapeTest.java**, **CircleTest.java**, **QuadrilateralTest.java** and **TriangleTest.java** from the class website. Import them into the project, modify the Build Path to add JUnit4, and test your classes with them.

Get your instructor's signature when all tests pass.

Exercise 1 - Get Instructor's Signature

2. Object Equality & Comparison

Equality tells whether 2 objects have the same representative data. Equality is enabled by the **equals** method implemented in Object (which is the super class of all Java classes). Equality is automatically used by array lists when using methods such as *contains* and *indexOf*.

Likewise, the interface **Comparable** is used to indicate whether an object precedes another when sorted using *Collections.sort* (if using an array list) or *Arrays.sort* (if using a plain array). In Java, comparison is enabled through the implementation of the **Comparable** interface.

The following exercise requires you to write classes that implement these features.

Exercise 2

As a party organizer, you will keep track of the people invited to a party and those who have RSVP-ed (of course, only those that have been invited can RSVP). To this end, you will write a *Party* and *Person* classes. A party object holds 2 array lists of persons: one for those invited and one for those that have RSVP-ed. Both lists can be sorted by people's names.

To take advantage of matching and sorting functionality in array lists, the Person class will override its **equals** method, and implement the **Comparable** interface (write it *Comparable<Person>* so that it compares against another person by default).

1. Create a **Lab04PartB** project.
2. Create a class **Person** implementing the Comparable interface (write it *Comparable<Person>*). The class has a private string field for the person's name, and the methods below:
   - A **constructor** receiving a string to initialize the person's name.
   - The method **getName**, which returns the name of the person.
   - The method **setName**, which receives a string and assigns it to the person's name.
   - The method **equals** (overriding the one from Object), which receives an object and identifies whether it is equal to this person, where 2 persons are equal if their names are equal.
   - The method **compareTo** (from Comparable), which receives a person and returns a number resulting from comparing the names of this and the received person; that is, the method returns
     i. a negative number, if the name of the person goes before (alphabetically) the name of the other person,
     ii. 0 if they are equal, or
     iii. a positive number if this person's name goes after the name of the other person.

     (Hint: since names are strings, and the String class implements Comparable, use String's own *compareTo* method to compare the names).
3. Create a class **Party** with 2 private ArrayLists for persons, one for those invited and one for those who have RSVP-ed. In addition, it should implement the methods below:
   - A default **constructor** initializing the lists.
   - The method **addInvited**, which receives a person to add to the list of invited people. Any person can be added to the list, as long as the person is not in the list already. To avoid people that could be added to the list and then change their names (a technique to get unexpected people to crash into your party), you should create a new person using the name of the person passed as a parameter and add this newly created person to the list.

- The method **getInvited**, which returns a copy of all people in the invited list. To avoid exposing the list to undesired external manipulations, you should return a copy of the list with newly created copies of each person in the list. (Why do we need to make a copy of the list if it is easier to just return a reference to the list? Well, if you pass a reference to anyone that asks you then anyone can change **your** list – perhaps removing your buddies and adding people you don't like. Say no to party crashers!)
- The method **addRSVP**, which receives a person and adds a copy of this person to the list of RSVP-ed people—but only if the person is in the invited list and has not RSVP already.
- The method **getRSVP**, which returns a copy list with copies of all people in the RSVP list.

5. Download the JUnit test files **PersonTest.java** and **PartyTest.java** from the class website and import them into the project. Test your classes using these test files.

Get your instructor's signature when all tests pass.

Exercise 2 - Get Instructor's Signature

Lab 4 Instructor's Signature Page

Student's Name: _____ Student ID _____

Exercise 1: _____ JUnit tests passed: _____

Exercise 2: _____ JUnit tests passed: _____