Lab walk-through #5

Topic **Implementing Abstract Data Types**

1. Introduction

Abstract data types (ADTs) are a fundamental tool for programming efficiently in an object-oriented programming language. Their principles are used as a basis for many data structures and also are essential for problem-solving. Implementing abstract data types properly is a skill and requires practice.

Whenever there is data of different types that logically belongs together, it is worthwhile to think about defining an ADT.

2. Lab Objectives

The objective of this lab is for the student to become familiar with abstract data types and how to build and use abstract data types in Java.

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer.
2. Completed Lab 1 from the previous week.
3. Read chapter 2 in your book on Abstract Data Types.
4. Have some basic knowledge about complex numbers.

4. Lab Exercise

The use of Abstract Data Types is essential when programming with object oriented languages like Java. There are many different types of ADTs in Java and they can be classified according to their function:

1. **Standard System ADTs** – can be used in any program automatically.
2. **Java ADT libraries** – can also be used in any program by use of an import statement.
3. **I/O ADTs** – used to manage streams of a program.
4. **Data oriented ADTs** – their purpose is to encapsulate data for management and security.
5. **Collection ADTs** – used to manipulate collections of data of the same type.
6. **Operations-oriented ADTs** – used to analyze algorithms.
7. **ADTs for graph algorithms** – used in graph-processing algorithms.

In this exercise, you will learn how to implement a data-oriented ADT (Geometric Object) using Java. Nevertheless, the same principles discussed for this example will apply for implementing other ADTs. We implement abstract data types using Java classes. We refer to a particular abstract data type as a data-type definition.

The first statement(s) in a data-type definition (class) are used to declare instance variables. These statements define data type values and hold the information that needs to be encapsulated by an instance of the class (object). Next, the constructor and instance methods follow. The instance methods perform operations on the instance variables. Instance methods may be public (specified in the API) or private (used to organize the computation and not available to clients). This data-type definition can contain many constructors and definitions of static methods. A method like `main()` as a unit-test is useful for debugging.

4.1. Implementing an Abstract Data Type using Java

This exercise will walk you through the example on page 77 of your Algorithms text book implementing Point2D ADT for geometric objects with a few changes. Before you start doing this section, in your Eclipse workspace create a project named 2XB3_Lab5 inside a package named cas.2XB3.lab5.wt. Add classes to this project as you walk through this instruction.

Every abstract data type implementation has three basic ingredients: API, clients and implementations.

Task1: Create an API for ADT

What is an Application Programming Interface (API) exactly? An API is a list of constructors and instance methods (operations), with an informal description of the effect of each. It contains a contract with all clients and is the starting point for developing any client or implementation of the ADT. The API of the ADT of a Point2D in two dimensions (as shown on page 77) looks like this:

public class	***Point2D***	
	Point2D(double x, double y)	*create a point*
double	x()	*x coordinate*
double	y()	*y coordinate*
double	r()	*radius (polar coordinates)*
double	theta()	*angle (polar coordinates)*
double	distTo(Point2D that)	*Euclidean distance from this point to* ***that***
void	toString()	*Returns a string representation*

In the first column of the API is the return type of each method. The middle column is the name of the relevant method and the third column describes what use the method has. You should write an API for each class you want to define.

Next you have to create the class in Eclipse. Give it the name Point2D as specified in the API. Then add the signatures of the functions, again as shown in this API. This gives you a skeleton of how your class will look like.

```
public class Point2D {
        Point2D(double x, double y)
        double x()
        double y()
        double r()
        double theta()
        double distTo(Point2D that)

        void toString()
}
```

Not only does the API allow you to implement the ADT, but you can also write client code without knowing the implementation details of an abstract data type. We can use an ADT in any .java file in the same directory, using an import statement or one that exists in the standard java libraries. Through encapsulation, we enable the implementation of client code at a higher level of abstraction.

Task 2. Implement ADT

Task 2.1. Defining Instance Variables

The first few lines of the implementation are the instance variables. These hold the information that needs to be encapsulated.

What information would be encapsulated in this case? We would need to encapsulate the Cartesian coordinate information. How would we encapsulate them? By making them private and (if needed and allowed) defining the appropriate accessor methods to alter the information safely.

Put the instance variables at the beginning of the class definition. Two instance variables are required, one for the x coordinate of the point and one for the y coordinate of the point. After adding them to your class, your code will look like this:

```
public class Point2D
{
        private final double x;
        private final double y;
  …
}
```

Task 2.2. Defining Constructors

Each implementation should have at least one constructor that establishes an object's identity. A constructor is like a static method but it can refer directly to an instance variable. It has no return value. A class can have multiple constructors with different signatures. The purpose of a constructor is to initialize the instance variables of a class. Any client code that uses an ADT or any java class would first need to call its constructor before any other functionality can be used.

To add a constructor for the Point2D ADT, create a function called Point2D with no return value and two parameters, the x and y values that are used to initialize the point. Inside the constructor, it is necessary to assign these arguments to the encapsulated information in the Point2D class. Code for the constructor should look like this after you are done:

```
public Point2D(double x, double y) {
        this.x = x;
        this.y = y;
    }
```

You should try and include a default constructor where possible. This is useful when no initial information is provided by the user of the ADT. A default constructor has no parameters. Add the following default constructor to your code:

```
public Point2D() {
        this.x = 0.0;
        this.y = 0.0;
    }
```

Task 2.3. Defining Accessor Methods

Accessor methods are extremely important for proper data encapsulation. Encapsulation is ensured by hiding the instance variables, but they are no use to anyone if they cannot be accessed by those authorized. Accessor methods help to solve this problem by allowing controlled access to such variables. Use them with caution, as incorrectly designed accessor methods are a security risk.

First, define the straightforward accessor methods that allow someone outside the class to get the x and y values encapsulated by the class. These two functions each return a double value and are named after the information that can be retrieved from them – the x and y coordinate values respectively. Inside each function, add the relevant return statement, as shown below.

```
public double x() { return x; }
public double y() { return y; }
```

The last two accessor methods are slightly more involved. To get the polar coordinates r and theta, you need to use the following mathematical formulas:

$$r = \sqrt{\ } (x2 + y2)$$
$$\theta = \tan\text{-}1 (y / x)$$

For this, you need to use the **sqrt** and the **atan2** functions from the Java Math library. When using library functions, you should read their documentation carefully so that you understand how to properly use them. Convert these formulas into code. In the end, the two accessor methods for the polar coordinates should look like this:

```
public double r() { return Math.sqrt(x*x + y*y); }
public double theta() { return Math.atan2(y, x); }
```

Note: You can find the entire API of the Math library online(or any Java library for that matter): http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html.

Task 2.4. Defining Instance Methods

Instance methods describe the behaviour of each object. We implement instance methods like any static methods in Java. Each instance method has a return type, a *signature* (which specifies its name and the types and names of its parameter variables), and a *body* (which consists of a sequence of statements, including a *return* statement that provides a value of the return type back to the client).

In this particular example, the first instance method, `distanceTo`, determines the Euclidean distance between two specified `Point2D` objects. With the right Math library functions, it is not hard to see how the implementation below represents the underlying mathematics. The second method is a simple `toString` override method that allows clients to see the encapsulated data.

Add them to your class to complete your implementation.

```
public double distanceTo(Point2D that)
{
        double dx = this.x - that.x;
        double dy = this.y - that.y;
        return Math.sqrt(dx*dx + dy*dy);
}



public String toString()
{
        return "(" + x + ", " + y + ")";
}
```

Now you should have completed your implementation of the ADT. It should contain one class, called Point2D which contains two instance variables, a constructor and four accessor methods. It should also contain two instance functions.

Task 3. Writing Client Code for ADT

Client code can be called in any public instance method in the implementation, replacing the parameter list with client values and returning the appropriate result, if any. The instance methods can access and perform operations on the instance variables. The object that calls the instance method is the one whose instance variables get modified.

In order to develop client code, you need to declare variables, create objects to hold data-type values and then provide access to the values for instance methods to operate on them.

Now, we will write possible client code for our `Point2D` example. Our client code will use our abstract data type as described below. Note the use of a constructor before any other methods are used. Add the following method to your class:

```
public static void main(String[] args)
{
        Point2D p = new Point2D();
        System.out.println("p  = " + p);
        System.out.println("   x     = " + p.x());
        System.out.println("   y     = " + p.y());
        System.out.println("   r     = " + p.r());
        System.out.println("   theta = " + p.theta());
        System.out.println();

        Point2D q = new Point2D(0.5, 0.5);
        System.out.println("q  = " + q);
        System.out.println("dist(p, q) = " + p.distanceTo(q));
}
```

Run your project. What do you see? If the ADT was implemented correctly, it should run smoothly and provide correct results. For this particular client code, you should see the following results:

```
p  = (0.0, 0.0)
  x     = 0.0
  y     = 0.0
  r     = 0.0
  theta = 0

q = (0.5, 0.5)
dist(p, q) = 0.7071
```

This section can be designed in many different ways, but if proper testing is to be done, all functionality and all edge cases must be tried. As you experiment with your implemented ADT, change this method to test out appropriate functionality of your implementation. Make sure that all functions work as intended and correct any errors as you go along. For instance, try this:

```
Point2D a = new Point2D();
System.out.println("a  = " + a);

Point2D b = new Point2D();
System.out.println("b  = " + b);
System.out.println("dist(a, b) = " + a.distanceTo(b));
```

What do you see? If your ADT works correctly, the distance between the two points must be `0.0` since they are the referring to the same point in Cartesian space. This is an example of an edge case for this implementation. Can you think of others to test your implementation properly?

Task 4. Input and Output files

Lastly, when testing extensively, you will need to give your program a large number of test cases to make sure it is working properly. You want to test your program without having to change the source file again and again in scenarios such as when you are required to send the results to someone in charge, or analyze the results later without having to run the program again. In these situations, it is very useful to be able to read your test cases from an external text file, and then write the results to another text file. Both files need to have a format that you stick to.

Create a text file in a separate folder (say, "*data*") in your project called "*input.txt*". In Java, we perform file input and output by using streams. Depending on what you want to do and what you prefer, there are many streams available. For file input, find online documentation on the "*Scanner*" class. For output, try using the "*PrintStream*" class. Let's look at how the Scanner class might be used:

```
Scanner input = new Scanner(new File("data/input.txt"));
while (input.hasNext()){
      String current = input.next();
}
input.close();
```

Objects of type `Scanner` are useful for breaking down formatted input into tokens and translating individual tokens according to their data type. You can also use buffered streams for more streamlined I/O. Label your output file as "*output.txt*". Then read in some of the test cases from your input file and write your results to your output file.

It is up to you to make sure you are comfortable with File I/O at the end of this exercise. It is essential to programming in Java and you will be asked to do this in many of your assignments.

Task 5. Submit your work
Once all tasks are completed, you should submit your Eclipse project. This must be done before leaving the lab. Follow the instructions below for submission:

- Include a .txt file named last_name_initials.txt in the root of the project containing on separate lines: Full name, student number, any design decisions/assumptions you feel need explanation or attention.
- After checking the accuracy and completeness of your project, right-click on the name of the project, select Export->General->Archive File.
- Ensure that just your project has a check-mark beside it, and select a path to export the project to. The filename of the zipped project must follow this format: *macID_Lab5.zip.* Check the option to save the file in .zip format. Click Finish to complete the export.
- Go to Avenue and upload your zipped project to 'Lab Walk-through 5 – Lab Section X)
- IMPORTANT: You MUST export the FULL Eclipse project. Individual files (e.g. java/class files) will NOT be accepted as a valid submission.

5. Further Practice Problems

For further practice, here are some suggested questions from your 2C03 textbook: 1.2.15, 1.2.16, 1.2.18.
Robert Sedgewick and Kevin Wayne.