LAB 2

OBJECT ORIENTED PROGRAMMING IN JAVA

A First Utility Class

PURPOSE OF LAB 02

In this lab you will learn how to define your own classes, with instance and class data members and instance methods.

- Create a directory called lab02_12345 where 12345 is replaced by your student Number
- Inside the directory, copy the files for each of the classes you created (only the .java).
- Create a Readme.txt which is a text file containing your name, student number and a brief description of the contents of your files

This is an individual lab. You may ask the lab tutor for help and you may consult with your neighbor if you are having difficulties.

Deliverables:
Q1)Rectangle.java, RectangleTest.java
Q2) Circle.java, CircleTest.java
Q3) Area.java
Q4) Box.java, BoxTest.java
Q5) Wood.java

Note: each file must contain you student name and number

Upload a zip file containing all your files with

Q1) FIRST UTILITY CLASS BASICS

Review of terminology

We have seen that, in Java, all application programs must be defined inside of a class that contains a `main` method. Let's categorize these classes as *driver classes* because they drive a program. We have also used some predefined classes that can be classified as *utility classes*, because they are utilized by programs and by other classes. Examples of utility classes that we have used from the Java API include classes that were instantiated, such as `javax.swing.JFrame`, `java.lang.String`, `java.util.Date`, `java.util.Scanner`, and `java.text.DecimalFormat`. These classes were used to create objects that were utilized in our programs. Java API classes such as `javax.swing.JOptionPane` and `java.lang.Math` can be used, by invoking class methods, without creating an object.

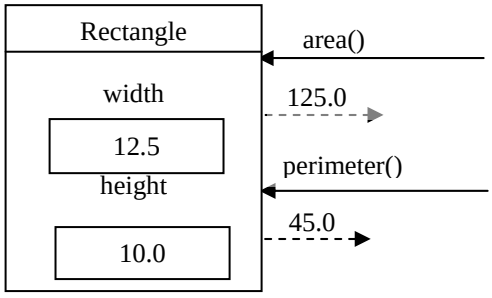
A class is a template that describes an object's data values and methods that work with this data. Collectively, the data and methods are referred to as the members of the class. There are two categories of data members, those that belong to an object and those that belong to the class. In addition, for each of these categories there are constant and variable data members. Since an object is an instance of a class, a variable data member that belongs to an object is called an *instance variable*. Each object that is created from a class has its own version of an instance variable. Conversely, a *class variable* belongs to the class. There is only one version of a class variable that is shared by all objects created from the class. Similarly, there are two types of method members, instance methods and class methods. An *instance method* must be invoked on an object. A *class method* does not need to be invoked on an object but, instead, can be invoked on the name of the class.

The Rectangle class

As a first example, we will define a `Rectangle` object. This will not be a rectangle that we can see, but rather a rectangle that can be used for calculating area and perimeter. The class could, for example, be used by a program that calculates the number of gallons of paint that are needed to paint a room.

A rectangle has a `height` and `width`, and given this data, the `area` and `perimeter` of a rectangle can be determined. Therefore, the class `Rectangle` should define data members `height` and `width`. Since each `Rectangle` object stores its own values for these variables, `height` and `width` are instance variables. The methods are the method's that determine the `area` and `perimeter` of a `Rectangle` object and are, therefore, instance methods. The Rectangle class contains only instance variables and instance methods. In later sessions and assignments, we will define utility classes that contain class data values and class methods.

Pictured above is a diagram of a Rectangle object showing the instance variables width and height and the methods area and perimeter. The ***state of the object*** are the values of its data members. Since the state of this object has been set, also illustrated are the values returned when the area and perimeter methods are invoked on the object.



Class definition

A typical class definition has the form

```
class className
{
   data member declarations
   method member definitions
}
```

Defining instance variables

We begin by declaring the instance variables, placing them at the top of the body of the class.

```
class Rectangle
{
    //instance variables
    double height, width;
}
```

Writing a method

While we have written many different `main` methods, we have never written a method other than `main`. We have, however, used many methods other than `main`. And, in learning how to invoke methods on objects, we have looked at some method headers. Recall that a method header has the form

optionalAccessModifiers returnType methodName(optionalParameterList)

To write the method that calculates and returns the area of a rectangle, we start with the method header.

- The methodName should indicate the method's purpose, so area is a good name.
- The parameterList represents information that the method needs to complete its task. We might conclude that the width and height should make up the parameter list. However, every instance method has access to the object's instance variables. Specifically, when the area method is invoked on a Rectangle object, it has access to the object's height and width variables. Since no additional information is needed, the method has no parameter list.
- The returnType is the data type of the value that is returned. Since height and width are of type double, the area of the rectangle is a double and, therefore, the return type is double.
- The only accessModifier needed is public. The modifier public allows any program that uses a Rectangle object to invoke the area method on a Rectangle object.

Therefore, the method header is `public double area()`, and the form of the method is

```
public double area()
{
   method body
}
```

The body of a method consists of statements that are executed when the method is invoked. The statements that make up the body of the `area` method must calculate the area and return the area.

A variable declared inside of a method is known as a *local variable* for the method and exists only during the execution of the method. These statements declare a local variable `theArea` and assign the area of the rectangle to `theArea`.

```
double theArea;
theArea = height * width;
```

Any method that returns a value must have a `return` statement. The term `return` is a Java reserved word. The statement that returns the area is:

```
return theArea;
```

Step 1: Place the code for the Rectangle class in a file called Rectangle.java.

```
class Rectangle
{
    //instance variables
    double height, width;

    public double area()
    {
       double theArea;
       theArea = height * width;
       return theArea;
    }
}
```

Compile the code. Try to execute the class file by executing the command

```
> java Rectangle
```

Record the error message.

_____

_____

_____

Testing the utility class

In order to use a Rectangle object, we must write a program. Therefore, we need two classes - the utility class Rectangle and a driver class RectangleTest that will be used to test our code. The two classes may be placed in the same file RectangleTest.java or they may be placed in two separate files, Rectangle.java and RectangleTest.java. We will choose the second option. Therefore, create a second file called Rectangle.java.

A program that declares and instantiates a Rectangle object

```
Rectangle myRect = new Rectangle();
```

can invoke the area method using the statement

```
double myArea = myRect.area();
```

which assigns to myArea the value returned by the area method.

Step 2: Enter and save this in a file called RectangleTest.java.

```
class RectangleTest
{
   public static void main(String[] args)
   {
      Rectangle myRect = new Rectangle();
      double theAreamy = myRect.area();
      System.out.println("My rectangle has area " + myArea);
   }
}
```

Compile the program `RectangleTest.java`. Execute the program and record the results.

_____

_____

Step 3: Currently, you can access the values of `width` and `height` directly by joining the variable to the name of the object using the dot operator. Add the following statements to the end of the `main` method.

```
System.out.println("Width is " + myRect.width);
System.out.println("Height is " + myRect.height);
```

Predict the output of the new program.

_____

_____

_____

Compile and execute the program. Record the results. Was your Step 2 prediction correct? If not, correct your answers.

Step 4: Modify the program by adding these statements at an appropriate place in the `main` method so that the area of `myRect` is no longer 0.

```
myRect.width = 2.0;
myRect.height = 3.3;
```

Compile and execute the program. Record the results.

_____

_____

_____

Access Modifier: private

Step 5: Being able to directly access the instance variables of an object (`Rectangle`) from an outside class (`RectangleTest`) is considered to be an inappropriate practice in object-oriented languages. To prevent this, the instance variables of a class should be modified by the access modifier `private`. Modify the `Rectangle` class by inserting the `private` modifier in the data member declaration statement:

```
private double width, height;
```

Compile the modified program. Record the compiler error messages.

_____

_____

_____

_____

The principle of making the data members of a class `private` and controlling access to the `private` data through the `public` methods is called *encapsulation*. Basically, this principle states that the integrity of an object is maintained by making the instance variables `private`,

```
private double width, height;
```

and, thereby, not allowing the user of the object to directly access the data, as in

```
System.out.println("Width is " + myRect.width);
System.out.println("Height is " + myRect.height);
```

or modify its data, as in

```
myRect.width = 2.0;
myRect.height = 3.3;
```

Accessor Methods

If the user of an object is to be allowed to access or get the value stored in a variable, an *accessor method* is needed. It is customary that the method is named `get` followed by the variable name. Therefore, we could choose to add the accessor methods `getWidth` and `getHeight` to our `Rectangle` class. Since the purpose of a `get` method is to allow the user to access the value stored in a data member, the method merely returns the required instance variable, a `double`. No information needs to be passed to a `get` method.

Step 6: In the `Rectangle` class, insert the code for the method `getWidth` which has no parameters and returns a `double`

```
public double getWidth()
{
   return width;
}
```

Compile the code. Then, insert a similar method to give the user, or client, access to the `height` of a `Rectangle`.

In the client class, any class that uses a `Rectangle` object, the illegal statement

```
System.out.println("Width is " + myRect.width);
```

should be modified to use the accessor method

```
System.out.println("Width is " + myRect.getWidth());
```

Modify the client class, `RectangleTest`, to correctly access the `width` and `height` of the `Rectangle` object.

Compile `RectangleTest.java`. **Note:** If `Rectangle.java` has not been compiled since changes to it were last made, it will be compiled when `RectangleTest` is compiled, because `RectangleTest` uses `Rectangle` objects.

Execute the `RectangleTest` program and record the results.

_____

_____

_____

_____

Mutator methods

If the user of an object is allowed to modify or set the value stored in a variable, a `set` method is needed. It is customary that the method is named `set` followed by the name of the variable. Therefore, we could choose to add the methods `setWidth` and `setHeight`. The `setWidth` method should have a formal parameter that receives the value to which the `width`, a variable of type `double`, should be set. For now, the name of this formal parameter should be anything except `width`, since `width` is the name of the instance variable. The method does not need to return a value, so the return type is `void`. Methods with return type `void` do not need a return statement but, a simple `return;` may be included.

The completed method should be added to the `Rectangle` class:

```
public void setWidth(double w)
{
   width = w;
}
```

The *state* of an object is defined by the values of its instance variables. Methods that set a variable to a new value change the state of the object or mutate the object. Therefore, set methods are called *mutator methods*.

In the program that uses a `Rectangle` object, the illegal statement

```
myRect.width = 2.0;
```

should be modified to use the mutator method

```
myRect.setWidth(2.0);
```

The statement above assigns the argument `2.0` to the method's formal parameter `w`. In the body of the method, the value stored in `w` is assigned to the instance variable `width`.

Step 7: Add the methods `setWidth` and `setHeight` to the `Rectangle` class. Make changes to `RectangleTest` to correctly use these methods. Your Rectangle class should now be:

```
class Rectangle
{
    private double width, height;

    public double area()
    {
        double theArea;
        theArea = height * width;
        return theArea;
    }

    public double getWidth()
    {
       return width;
    }

    public double getHeight()
    {
        return height;
    }

    public double setWidth(double w)
    {
        return width;
    }

    public void setHeight(double h)
    {
        height = h;
    }
 }
```

And, your RectangleTest class should now be:

```
class RectangleTest
{
    public static void main(String[] args)
    {
        Rectangle myRect = new Rectangle();

        myRect.setWidth(2.0);
        myRect.setHeight(3.3);
        double theArea = myRect.area();

        System.out.println("Width is " + myRect.getWidth());
        System.out.println("Height is " + myRect.getHeight());
        System.out.println("My rectangle has area " + theArea);
    }
}
```

Compile RectangleTest. Execute the program and record the results.

_____

_____

_____

Writing a Constructor

The second way to set the values of the instance variables is to set them at the time the object is created. The RectangleTest statement

```
Rectangle myRect = new Rectangle();
```

uses the constructor Rectangle(), a default constructor. If a class does not define its own constructor, a default constructor , which has no parameters and sets all data members to the equivalent of zero, is automatically provided. To set the values of the instance variables when the object is created, we need to write a Rectangle constructor that has two parameters of type double that are used to initialize the Rectangle's width and height. A constructor is a special type of method. The form of a constructor is

```
public className( optionalParameterList )
{
   body
}
```

A constructor must have the same name as the class name. Therefore, a constructor used to construct a Rectangle object, must be named Rectangle. We say that a constructor is a special type of method because it does not have a return type and because it can only be used in conjunction with the new operator. A constructor that initializes the height and width of a Rectangle object would take the form

```
public Rectangle(double w, double h)
{
    width = w;
    height = h;
}
```

The statement

```
Rectangle myRect = new Rectangle(2.0, 3.3);
```

constructs a new Rectangle object with width equal to 2.0 and height equal to 3.3. When executed, the argument 2.0 is assigned to the parameter w and the argument 3.3 is assigned to the parameter h. In the body of the constructor, the

value stored in `w` is assigned to `width` and the value stored in `h` is assigned to `height`. The order in which the arguments are assigned to the parameters is determined by the order in which they are passed. Therefore, the statement

```
Rectangle myRect = new Rectangle(3.3, 2.0);
```

assigns `3.3` to `w` and `2.0` to `h`.

Once we have a constructor that creates and initializes an object, the `set` methods are no longer needed, but may still be included. Whether to provide the user of an object with `get` and `set` methods is a design decision. Very often, `get` methods are provided to give the client access to an instance value, but `set` methods that allow the client to mutate the object are not.

Step 8: Modify the `Rectangle` class by adding the constructor, placing it after the declaration of the instance variables, and before the definitions of the existing methods. This location is not mandatory, but it makes the code more readable.

Also, do the following to modify the class `RectangleTest` class

1. Comment out the two statements in `main` that invoke the `set` methods.
2. Change the statement that creates the `Rectangle` object from

```
Rectangle myRect = new Rectangle();
```

to

```
Rectangle myRect = new Rectangle(12.5, 10);
```

Compile and execute the modified program. Record the results.

_____

_____

_____

_____

Overloaded methods

Overloaded methods are methods from the same class that have the same name but different parameter lists. That is, a class may define two methods with the same name as long as the signatures of the methods are different. For example, the `String` class contains the methods

```
public String substring(int beginIndex, int endIndex)
public String substring(int index)
```

Constructors may also be overloaded. When the constructor

```
public Rectangle(double w, double h)
```

was added to the `Rectangle` class, the default constructor `public Rectangle()`became inaccessible. It is a good practice to always include a constructor that has no parameters. In this case, we should add

```
public Rectangle()
```

to the `Rectangle` class. This constructor could initialize both instance variables to `0`, or to any other default value of the programmer's choosing. Assigning 1 as the default dimension would be one such choice.

Step 9: Make two additions to the `Rectangle` class and modify `RectangleTest`.

A. To store the default dimension, introduce another data member that is a class constant. Place at the beginning of the class definition the statement

```
public static final double DEFAULT_DIMENSION = 1;
```

and answer these questions:

1. What modifier makes this data member a class constant instead of an instance constant?

_____

_____

2. What modifier makes this data member a constant instead of a variable?

_____

_____

3. Why is the integrity of the class not compromised by modifying the data member with public?

_____

_____

B. Modify the `Rectangle` class by adding a second constructor, placing it either before or after the first constructor.

```
public Rectangle()
{
    width = DEFAULT_DIMENSION;
    height = DEFAULT_DIMENSION;
}
```

C. Add code to `RectangleTest` to test the new constructor by creating a second `Rectangle` object named `yourRect` and printing its dimensions and area.

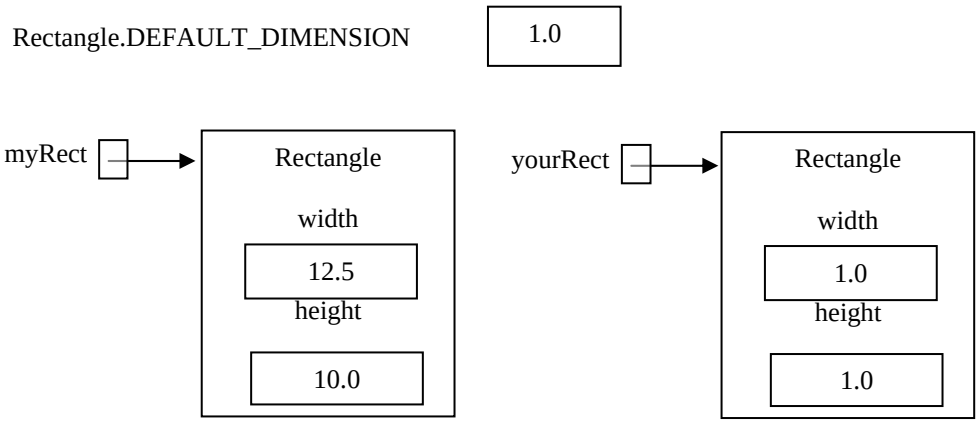
Compile and execute the program. Record the results.

_____

_____

_____

_____



A state of memory diagram for the current RectangleTest program

Writing a toString method

Step 10: We know that we can print `String` and numeric values. What happens if we print an object? Add this statement to the end of the main method in RectangleTest.

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

Compile and execute the program. Record the new print results.

_____

_____

A utility class should define a method with header `public String toString()` that returns a `String` representation of the object. The `String` that is returned is determined by the programmer. Note that the method does not print a `String`, it returns the `String`. A `toString` method for the `Rectangle` class could be

```
public String toString()
{
   String s;
   s = "Rectangle: dimensions " + width + " x " + height;
   return s;
}
```

Step 11: Modify the two classes

A. Add the `toString` method to the `Rectangle` class.

B. Add the print statement to the end of the `main` method in `RectangleTest`

```
System.out.println(myRect.toString());
```

that calls the `toString` method and prints the returned `String`. You should now have two print statements

```
System.out.println(myRect);
System.out.println(myRect.toString());
```

Compile and execute the program. Record the results of these two print statements.

_____

_____

_____

Draw a conclusion: What is printed when an object is printed if the class defines a `toString` method?

_____

_____

_____

Completing the Rectangle class

Now, complete the Rectangle class by

1. Rewriting the area method
2. Adding a method to find the perimeter of a Rectangle object
3. Adding a third constructor

There is usually more than one way to write a method. The area method is such a method. This second version combines the declaration and initialization statements of the local variable theArea.

```
public double area()
{
   double theArea = height * width;
   return theArea;
}
```

In the third version, the local variable theArea is omitted. The calculation of the area can be done in the return statement. The parentheses surrounding the calculation are optional.

```
public double area()
{
   return (height * width);
}
```

Step 12: Replace the body of the existing area method with one of the new versions. Compile the revised Rectangle class and execute RectangleTest to ensure that the new version works.

Step 13: A. Modify the Rectangle class by adding a method with the header

```
public double perimeter()
```

that calculates and returns the perimeter of the rectangle.

B. Modify the RectangleTest class by adding statements that print the perimeters of the two existing objects.

Compile the revised RectangleTest and run it to ensure that the new version works.

Step 14: A. Add a third constructor to the Rectangle class with the header

```
public Rectangle(double side)
```

that initializes both instance variables to side, i.e. the rectangle is a square.

B. Also, make the following changes at the end of the main method in the RectangleTest class:

1. Declare and create another Rectangle object using the new constructor.
2. Add statements that find and print the dimensions, area, and perimeter of the third Rectangle object.

Compile the revised RectangleTest and run it to ensure that the new version works.

Q2) THE CIRCLE CLASS

Write a utility class Circle and a driver class CircleTest test each constructor and method in the class. Recall that the Math class defines the constant public static final double PI.

A Circle object should have one ***instance variable***: double radius and one ***class constant*** that defines the default radius of 1.

Include two ***constructors***:

- public Circle(double r) initializes the radius to r
- public Circle() initializes the radius to the default radius

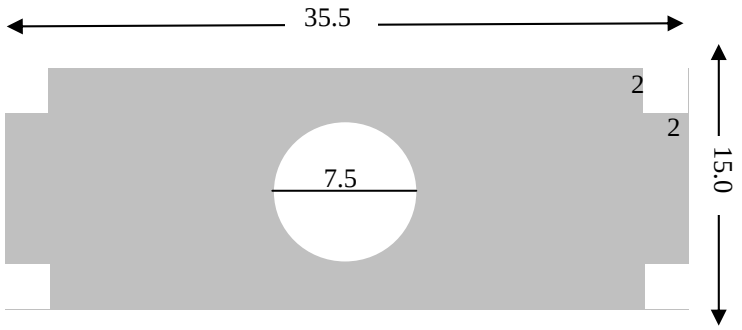
Include the following ***methods***:

- public double area() returns the area (πr^2)of this Circle
- public double circumference() returns the circumference ($2\pi r$) of this Circle
- public double diameter()
- public String toString() returns a String representation of this Circle such as

  "Circle with radius 2.5"
- public void setRadius(double r) sets the radius to r
- public double getRadius() returns the radius of this Circle

Q3) THE AREA CLASS

Write a program Area.java that uses both Rectangle and Circle objects to find the area of the shaded region. All of the cutout corners are squares with side 2. The area should be printed, rounded to one decimal place.



Q4) THE BOX CLASS

Write a utility class Box and a driver class, BoxTest.java , used to test each constructor and method in the class. A Box object should have three instance variables : double height, width, depth;

Include three ***constructors***:

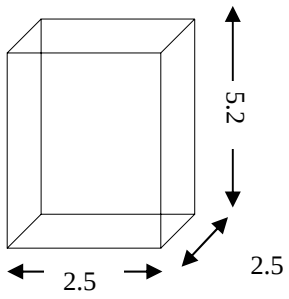
- public Box(double w, double h, double d) initializes the three instance variables
- public Box() initializes all instance variables to 1
- public Box(double side) initializes all instance variables to side

Include the ***methods***:

- public double volume() returns the volume of this Box
- public int surfaceArea() returns the surface area of this Box
- public String toString() returns a String representation of this Box such as "Box with dimensions 4.3 x 6.5 x 9.0"
- public double diagonalLength() returns the length of the diagonal, the square root of the sum of the squares of each dimension, of this Box.
- Three accessor methods (get methods), one for each instance variable
- Three mutator methods (set methods), one for each instance variable

Q5) THE WOOD CLASS

4.4 Write a program Wood.java. A box is to be constructed out of wood with a double thickness of wood on the square bottom. Use the Box and Rectangle classes to determine the volume of the box and the surface area of the wood needed to construct the box.



Optional class:

Write a utility class Name and a driver class to test each constructor and method in the Name class.

Each Name object should have three ***instance variables***: String first, middle, last;

Include two ***constructors***:

- public Name(String f, String m, String lt) that initializes the three instance variables.
- public Name(String wholeName) that is split to initialize the three instance variables.

Include the ***methods***: (For the examples that follow assume the name is John Ty Smith.)

- public String intials() returns a String containing the three initials of the name. For our example: "J. T. S." is returned.
- public String toString() returns a String containing the first name, middle initial and last name. For our example: "John T. Smith" is returned.
- public String toFullString() returns a String containing the first, middle and last names. For our example: "John Ty Smith" is returned.
- public String toLastString() returns a String containing the last name, a comma, and the first name. For our example: "Smith, John" is returned.
- public String toUpperCase() returns a String containing the full name in upper case letters. For our example: "JOHN TY SMITH" is returned.