

---

# **Fundamentals of Defining a Utility Class**

In this laboratory session you will:

1. Learn to define a class that describes an object
2. Learn to write methods
3. Investigate encapsulation
4. Write overloaded constructors
5. Investigate the `Math` class

## Writing a First Utility Class - Part 1 - The Basics

In the previous lab sessions, we have seen that, in Java, all programs must be defined inside of a class. For the purpose of discussion, let's categorize these classes as *driver classes* because they drive a program. We have also used some predefined classes from the Java API to create objects in our programs. For example, the classes `java.util.Scanner`, `java.util.Random`, `java.lang.String`, and `java.text.DecimalFormat` define objects that can be utilized in our programs. Let's categorize such classes as *utility classes*. In this session, you will learn to write your own utility classes.

A class is a template for an object that describes the object's characteristics and behaviors. The characteristics of an object are stored in variables. These variables are called the *data members* of the class. The behaviors of an object are the methods that can be invoked on the object. These methods are called the *method members* of the class. Collectively, the data and methods are referred to as the members of the class.

There are two kinds of data members, instance variables and class variables. Since an object is an instance of a class, a variable that belongs to an object is called an *instance variable*. That is, each object created from the class has its own version of an instance variable. Conversely, a *class variable* belongs to the class. There is only one version of the class variable that is shared by all objects created from the class. For example, a `BankAccount` class could have data members `owner`, `accountNumber` and `balance`. These would be instance variables since each `BankAccount` object has unique values for these variables. The `BankAccount` objects could share the class variable storing the bank's `roundingNumber`.

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 utility classes that we write will contain only instance variables and instance methods. Later, we will look at the predefined `Math` class that contains class variables and class methods.

A typical class definition has the form

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

As a first example, let's define a `Rectangle` object. A rectangle has a length and width, and given this data, the area and perimeter of a rectangle can be determined. Therefore, the class `Rectangle` should define data members representing an object's length and width. Since each `Rectangle` object stores its own values for these variables, length and width are instance variables. The instance methods are the method's that determine the area and perimeter of a `Rectangle` object. All of this, and more, can be described in the class `Rectangle`.

```
class Rectangle
{
    double length, width;

    definition of area method
    definition of perimeter method
}
```

## 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 the 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 `length` 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 `length` 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 `length` and `width` are of type `double`, the area of the rectangle and therefore the return type, is `double`.
- The only *accessModifier* needed is `public`. This allows any program that uses a `Rectangle` object to invoke the `area` method on the object.

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

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

The body of a method consists of statements that are executed when the method is invoked. These statements can be any type of statement including declaration, assignment, selection, or iterative statements. 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 two statements declare a local variable `a` and, assign the area of the rectangle to `a`.

```
double a;
a = length * 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 a;
```

Putting this together we have

```
public double area()
{
    double a;
    a = length * width;
    return a;
}
```

A program that uses a `Rectangle` object called `myRect` and a double variable called `theArea` can invoke the `area` method using the statement

```
theArea = myRect.area();
```

which assigns to `theArea` the value returned by the `area` method.

### Experiment 7.1

In order to use a `Rectangle` object, we must write a program. Therefore, we need two classes - the utility class `Rectangle` and the driver class `J07E01`. The two classes may be placed in the same file `J07E01.java` or they may be placed in two separate files, `Rectangle.java` and `J07E01.java`. For this example we will choose the first option. The class `J07E01` contains a `main` method in which a `Rectangle` object is created, the `area` method is invoked, and the area is printed. Before compiling and executing the program, carefully examine both classes that are stored in the file `J07E01.java`.

```
class J07E01
{
    public static void main(String[] args)
    {
        Rectangle myRect = new Rectangle();
        double theArea;
        theArea = myRect.area();
        System.out.println("My rectangle has area " + theArea);
    }
} //end of class J07E01

class Rectangle
{
    double width, length;
    public double area()
    {
        double a;
        a = length * width;
        return a;
    }
} //end of class Rectangle
```

**Step 1.** Compile and execute the program `J07E01.java`. Record the results.

---



---



---



---



---



---



---



---

**Step 2.** Make a prediction: What are the values of `width` and `height` in the `myRect` object?

---

---

---

---

---

---

---

---

---

---

**Step 3.** Currently, you can access the values of `width` and `length` 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("Length is " + myRect.length);
```

Compile and execute the program. Record the results. Was your Step 2 prediction correct?

---

---

---

---

---

---

---

---

---

---

**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.length = 3.3;
```

Compile and execute the program. Record the results.

---

---

---

---

---

---

---

---

---

---

**Step 5.** Being able to directly access the instance variables of an object (`Rectangle`) from an outside class (`J07E01`) 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, length;
```

Compile the modified program. Record the compiler error messages.

---



---



---



---



---



---



---



---

## Writing a First Utility Class - Part 2 - Writing `get` and `set` Methods

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, length;
```

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("Length is " + myRect.length);
```

or modify its data, as in

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

If the user of an object is to be allowed to access or get the value stored in a variable, a `get` method is needed. It is customary that the method is named `get` followed by the variable name. Therefore, we could choose to add the methods `getWidth` and `getLength` 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. Here is the method `getWidth`:

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

In the program 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());
```

(An *accessor method* is a method used to access data within an object.)

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 `setLength`. The `setWidth` method should have a formal parameter that receives the value to which the width, a variable of type `double`, should be set. The name of this formal parameter can be anything except `width`, since `width` is the name of the instance variable. The method does not need to return a value and therefore has return type `void`. Methods with return type `void` do not need a return statement. The following method could 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 `width`.

## Experiment 7.2

**Step 1.** Compile and execute the program `J07E02.java` that contains the highlighted, suggested modifications. Record the results.

```
class J07E02
{
    public static void main(String[] args)
    {
        Rectangle myRect = new Rectangle();
        double theArea;

        myRect.setWidth(2.0);
        theArea = myRect.area();
        System.out.println("My rectangle has area " + theArea);
        System.out.println("Width is " + myRect.getWidth());
    }
}
```

```
class Rectangle
{
    private double width, length;
    public double area()
    {
        double a;
        a = length * width;
        return a;
    }

    public double getWidth()
    {
        return width;
    }

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

---

---

---

---

---

---

---

**Step 2.** Of course, since we have not yet included a method to set the length of the Rectangle, the area is still 0.0. Modify the Rectangle class by adding the setLength method. And, modify the J07E02 class by adding a statement that sets the length to 3.3. Record your modifications.

---

---

---

---

---

---

---

---

---

---

---

---



## Writing a First Utility Class - Part 3 - 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. To do this, we need to write a constructor. A *constructor* is used to create an object and to initialize the object's instance variables. We were able to construct a `Rectangle` object in `J07E01.java` and `J07E02.java` using the default constructor that is automatically provided if we do not write our own constructor. This default constructor which has no formal parameters initializes all data members to the equivalent of zero. 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 `length` and `width` of a `Rectangle` object would take the form

```
public Rectangle(double ln, double w)
{
    length = ln;
    width = w;
}
```

The statement

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

constructs a new `Rectangle` object with `length` equal to 3.3 and `width` equal to 2.0. When executed, the argument 3.3 is assigned to the parameter `ln` and the argument 2.0 is assigned to the parameter `w`. In the body of the constructor, the value stored in `ln` is assigned to `length` and the value stored in `w` is assigned to `width`. 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(2.0, 3.3);
```

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

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.

### Experiment 7.2 continued

**Step 6.** Modify the `Rectangle` class in `J07E02.java` by adding a constructor, placing it after the declaration of the instance variables, and before the definitions of the existing methods. Also, do the following to modify the class `J07E02` class

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

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

to

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

Compile and execute the modified program. Record the results.

---



---



---



---



---



---



---

### Writing a First Utility Class - Part 4 - More Constructors and Methods

The `get` and `set` methods always follow the same pattern. However, there is usually more than one way to write most methods. The `area` method is such a method. Here are two more versions of the `area` method that could replace the existing method. In this version, the declaration and initialization statements are combined.

```
public double area()
{
    double a = length * width;
    return a;
}
```

In the third version, the local variable `a` is omitted. The calculation of the area can be done in the return statement. The parentheses shown surrounding the calculation may be omitted.

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

Recall that *overloaded methods* are methods from the same class that have the same name but different parameter lists. For example, the `String` class contains the methods

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

Constructors may also be overloaded. When the constructor

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

was added to the `Rectangle` class, the default constructor `public Rectangle()` became inaccessible. Therefore, we should add a constructor that has no parameters:

```
public Rectangle()
```

This constructor could initialize both instance variables to 0, or to any other values of the programmer's choosing. Assigning both instance variables a value of 1 might be one such choice.

### Experiment 7.3

**Step 1.** The file J07E03.java contains the updated classes.

```
class J07E03
{
    public static void main(String[] args)
    {
        Rectangle myRect = new Rectangle(2.3, 5.7);
        double theArea;
        theArea = myRect.area();
        System.out.println("My rectangle has area " + theArea);
    }
}

class Rectangle
{
    private double width, length;

    public Rectangle(double len, double wid)
    {
        length = len;
        width = wid;
    }

    public double area()
    {
        double a;
        a = length * width;
        return a;
    }

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

    public void setLength(double ln)
    {
        length = ln;
    }

    public double getWidth()
    {
        return width;
    }

    public double getLength()
    {
        return length;
    }
}
```

Compile and execute the program. Record the results.











**Step 10.** Compile and execute the program. Record the results.

---



---



---



---



---



---



---



---

### **Class Variables and Class Methods in the Integer and Math Classes**

Recall that a class variable belongs to the class, not to an object of the class. And, an object is not needed to invoke a class method, which can be invoked on the name of the class. The `java.lang.Integer` class has both class variables and class methods. In Session 3, we printed the maximum `int` value, which is stored in the `Integer` class as

```
public static final MAX_VALUE;
```

The modifier `static` states that this data member is a class variable. The modifier `final` states that the assigned value can never be changed and that the variable is therefore a constant. Since a constant can never be changed it is safely modified by `public`. To access the constant we wrote

```
System.out.println("The maximum int value is " + Integer.MAX_VALUE);
```

The `Integer` method that we have already used has header

```
public static int parseInt(String s)
```

The modified `static` states that the method is a class method. To use this method we have written statements such as

```
num = Integer.parseInt(line);
```

where `num` is of type `int` and `line` is of type `String`.

The `Math` class which is also defined in the `java.lang` package contains class variables representing  $\pi$  (`Math.PI`) and  $e$ , the natural number. To access these constants we use `Math.PI` and `Math.E`, respectively. Methods in the `Math` class include

```
public static double sqrt(double number)
```

that returns the square root of `number`,

```
public static double pow(double base, double exp)
```

that returns `base` raised to the power of `exp` (`baseexp`), and

```
public static long round(double number)
```

that returns `number` rounded to the nearest integer of type `long`







## Optional Post-Laboratory Problems

**7.1.** Write a utility class `Circle` and a driver class used to test each constructor and method in the `Circle` class.

A `Circle` object should have one instance variable: `double radius`.

Include two constructors:

```
public Circle(double r) initializes the radius to r
public Circle() initializes the radius to 1
```

Include the following methods:

```
public double area() returns the area ( $\pi r^2$ ) of this Circle
public double circumference() returns the circumference ( $2\pi r$ ) of this Circle
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
```

**7.2.** Write a utility class `Box` and a driver class used to test each constructor and method in the `Box` class.

A `Box` object should have three instance variables: `int length`, `width`, `height`.

Include three constructors:

```
public Box(int ln, int w, int h) initializes the three instance variables
public Box() initializes all instance variables to 1
public Box(int side) initializes all instance variables to side
```

Include the following methods:

```
public int 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 x 6 x 9"
public double diagonalLength() returns the length of the diagonal of this Box,
    the square root of the sum of the squares of each dimension.
Three accessor methods (get methods), one for each instance variable
Three mutator methods (set methods), one for each instance variable
```

**7.3.** 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 `first`, `middle`, `last` of type `String`. Include one constructor:

```
public Name(String f, String m, String lt)
```

that initializes the three instance variables. Include the following methods (For the examples that follow assume the name is John Thomas Smith.):

```
public String initials() 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 Thomas 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 THOMAS SMITH" is returned.

- 7.4.** Write an application that uses a loop to print a chart of the numbers 1-10 and their square roots.
- 7.5.** Write an application that uses a loop to print a chart of the numbers 1-10 and the numbers squared and cubed.