

Objectives

After completing the lab you should be able to

1. Explain the basic structure of a Java class.
2. Use object constructors.
3. Understand how instance variables are used by a class to define its state.
4. Define, use, and write methods.
5. Create a client program to use a Java object.
6. Create your own simple class and test it.

References: The following references will be used throughout the lab.

- *Course website:* <http://www.eg.bucknell.edu/~csci203/>
- *Java API website:* <http://java.sun.com/javase/6/docs/api/>
- Chapter 3 of Big Java

Preparation

Prepare for the lab by completing the following **To-Do** box.

To-Do:	Lab Preparation
1. Start Eclipse. 2. Create a new Java project named <code>lab05-xyz01</code> , where <code>xyz01</code> is your login name. 3. Import the file <code>Point.java</code> from <code>~csci203/2009-fall/student/labs/lab05</code> 4. Create the <code>readme.txt</code> file for this lab, insert the banner file with proper updates.	

Introduction

Classes are a key feature of the Java language. The Java class is the structure within which a program implements algorithms. It is also the mechanism for Java to organize data. In this lab you will explore the basic components within a class. You will also learn a way to test components of a class.

Studying the `Point` Class

Open the `Point.java` file that you imported and study it carefully. The class `Point` has all the components of a typical Java class.

1. A heading for the class stating the class's name.
2. A list of instance variables that establish the state of an object — notice that these variables are declared `private`.
3. One or more constructors that initialize the state.
4. Methods to modify the state of an object (mutators).
5. Methods to provide access to the state of an object (accessors).
6. A `toString` method that returns a `String` object used to print the state of the class.
7. A `main` method that can be used to test the class.

There are some details to note in each of the seven pieces of a Java class. We'll discuss them in details. Some of the discussions will be followed by an exercise or two.

A heading for the class stating the class's name

The definition of a class consists of a class heading followed by a *body* enclosed in a pair of braces `{}`. Inside the body is a collection of instance variable, constructors and methods. *Instance variables* are also known as fields or state variables.

A list of private instance variables that store the state of the objects

It is important to understand that when each `Point` object is created (we often say *instantiated*), it gets its own set of instance variables.

One or more constructors that initialize the state

One can easily tell a method from a constructor. A constructor *always* has the same name as the class. A constructor *never* has a return type, not even `void`. A class may have several constructors as long as the types or number of parameters differ. We say they have different *signatures*. The signature of a method is the method name, and the number of parameters and their types. For example, the signature of the two parameter `Point` constructor is

```
Point(double x, double y)
```

The `Point` constructor initializes the two instance variables to zero.

Exercise 1: Class Constructors

At this point you should make the following modifications to the `Point` class method `main`. Add two new declarations for `Point` variables (`p1` and `p2`, for example). Then instantiate the new `Point` objects that will represent different points with different coordinates, e.g., $(2, 3)$ and $(4, 5)$. Use the two parameter constructor to do this. Have the `main` method display the values for the three point values at the end of the method. This illustrates that each object has different values assigned for its instance variables.

Execute the program. Copy and paste the output into your `readme.txt` file.

It is not required that a class have a declared constructor. When there is no declared constructor, Java provides a *default* constructor that takes no parameters.

Methods to modify the state of objects

Some methods change one or more of the instance variables. A method that changes the state of an object is called a *mutator*. A mutator whose sole purpose is to change a single instance variables is called a *setter*. By convention, these mutators have names that begin with “set”.

Exercise 2: Mutator Methods

Write a new method named `setLocation` with the following description.

```
/**
 * Sets the location of the point to the specified location.
 *
 * @param x
 *         the x coordinate of the new location
 * @param y
 *         the y coordinate of the new location
```

```
*/  
public void setLocation(double x, double y)
```

Add code to the main method of the `Point` class to test the new method. Execute the program and copy the output to your `readme.txt` file.

Methods to provide access to the current state of objects

Some methods access the current state or value of an instance variable. By convention such methods have their names start with “get.” A method that provides access to an instance variable is called an *accessor* or *getter*.

A `toString` method that returns a `String` object used to print the state

When using the method `System.out.print` to print a message, Java uses the `toString` method to print the state of the object even if only the name of the object is used. For example, the following `println` method will call the `toString` method by default.

```
System.out.println(p1);
```

Exercise 3: Use of the `toString` method

Modify the method `main` by removing the `toString` calls from each `System.out` statement. Execute the program to show that the `toString` method is called by default. Put the output in your `readme.txt` file.

A main method to test the class

An important use of the main method is to insert code to test the methods in the class. We started the test code using `System.out.println` messages to explain what is being tested. One should be able to read the output and understand the test cases.

Exercise 4: Computing distance to a specified point

Determining the distance between points is a common operation on points. Add a method to the `Point` class with the following description.

```
/**  
 * Returns the distance from this point to a specified point.  
 *  
 * @param aPoint
```

```
*           the specified point to be measured against this point
* @return the distance between this point and the specified point.
*/
public double distance(Point aPoint)
```

When we refer to “this point”, we mean the receiver of the message. So, in the code for the method, you will use the values of `xCoord` and `yCoord` of *this instance of the object* along with the *x-coordinate* and *y-coordinate* you retrieve from the parameter `aPoint`.

Here are some hints for completing this method.

1. The distance between two points x and y is calculated as follows

$$\text{distance} = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

2. You will need to use the library method `Math.sqrt` to compute the square root.
3. To compute the difference between the x values, you will need to use the `getXCoord` method to get the x value of the parameter.

```
double xDiff = xCoord - aPoint.getXCoord();
```

You will need to use a similar technique to compute the y difference.

4. You won't need a square method since you can just multiply `xDiff` by itself to compute the square.

Add code to the `main` method to test your new method. When testing, you really should know the result *before* the program execution. For example, in the following test case, you should know the result is 5.0 if you have a point at (0,0) and the second point at (3,4). Thus, the following Java statements in your test program will make perfect sense.

```
Point p1 = new Point(0, 0);
Point p2 = new Point(3, 4);
System.out.println("Testing the distance method");
System.out.println("Distance from " + p1 + " to " + p2 + ": "
    + p1.distance(p2));
System.out.println("Expected: 5.0");
```

You expect the output from the program to be

```
Testing the distance method
Distance from (0.0, 0.0) to (3.0, 4.0): 5.0
Expected: 5.0
```

Test the above case. Execute the program with one more test case of your own to test the `distance` method.

Put the output from each of your tests that show the method works into your `readme.txt` file.

Using an Object through a Client Program

Objects are created for application programs to use. For example, one would imagine that a `Point` object can be used in various geometry related applications. A `String` object can be used in various text processing related applications. An application program that uses existing objects is called a *client program*. In this segment of the lab, you are to write an application program to make use of `Point` objects.

Exercise 5: Implementing a Distance Finder

Create a new class called `DistanceFinder`. (Make sure that the `src` directory is highlighted when you do this.) The new program is called a *client program* which will make use of the class `Point`. The purpose of this program is to create a sequence of points and compute the distance that would be traveled starting at the first, continuing to the last, and coming back to the first point.

The specifications for the program are as follows. The program will create three points. Then, it will compute the distance between the points and print it. Place all the code in the `main` method of `DistanceFinder` class.

Here is a trace of what your program should produce.

```
Point 1: (0.0, 0.0)
Point 2: (0.0, 4.0)
Point 3: (3.0, 0.0)
The total distance is 12.0
```

Place the output from two runs of your program into your `readme.txt` file. One of the runs must be for the three point sequence above.

Creating Your Own Class

Exercise 6: Implement a Savings Account Class

Write a class `SavingsAccount` that is similar to the `BankAccount` class in your book. It will start with the same constructors, methods and instance variable with the following changes and additions.

- Add an instance variable `interestRate`. It will be stored as a *percentage*.
- The constructor that has no parameters now needs to initialize the interest rate to zero.
- Replace the constructor that accepts an initial balance with a constructor that sets both the initial balance and the interest rate.
- Supply a method `addInterest` (with no parameters) that adds interest to the account balance. It will convert the interest rate to a decimal, calculate the interest and add it to the balance.

Write a `SavingsAccountTester` class that constructs a savings account with an initial balance of \$1,000 and an interest rate of 10%. Then apply the `addInterest` method five times and print the resulting balance. Also print the expected result.

Balance: 1610.51

Expected: 1610.51

Run your program and place a copy of the output in your `readme.txt` file.

What to Submit

Your `readme.txt` file should include the results from all the exercises above (exercise 1 through exercise 6).

1. Results from exercise 1
2. Results from exercise 2
3. Results from exercise 3
4. Results from exercise 4 (test cases)
5. Results from running your exercise 5 program
6. Results from running your exercise 6 program

Make sure your `readme.txt` file is in your lab directory.

Drag your lab directory to the CSCI 203 lab drop box: [CSCI203 Lab drop_box](#)