Carleton University
Department of Systems and Computer Engineering
SYSC 2004 - Object-Oriented Software Development - Winter 2015

Lab 3 - Collaborating Objects

Objective

In this lab, you will learn how object-oriented software is composed of collaborating objects, in which methods defined in one class call methods on objects that are instances of other classes. You'll gain further experience with interactive testing (using BlueJ's object bench and object inspectors) and automated unit testing (using JUnit test classes).

Attendance/Demo

To receive credit for this lab, you must demonstrate your work. **Also, you must submit your lab work to cuLearn by the end of the lab period**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will review the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before Lab 4.**

Background Reading

Objects First with Java, Chapter 3.

Getting Started

Step 1: Create a folder named Lab 3.

Step 2: Download lines.zip from cuLearn and move it to your Lab 3 folder.

Step 3: Right-click on lines.zip and select Extract All... to extract all the files into a project folder named lines.

Step 4: Open the lines folder. Double-click on the BlueJ project file, which is named package. This will launch BlueJ and open the *lines* project.

A class diagram containing four classes (Point, PointTest , LineSegment and LineSegmentTest) will appear in the BlueJ main window.

Exercise 1 - Developing a Class that Models 2-D Points

Step 1: You have been provided with a very incomplete implementation of class Point. Double-click on the Point class icon. The Java source for the class will appear in an editor window.

Step 2: Read the documentation for class Point. Define two private fields (instance variables) of type double, named x and y. **Important: Do not use different names for the fields. Do not define additional fields. The fields' visibility must be private, not public.**

Step 3: Read the Javadoc comments for the two constructors and the getX and getY accessor methods. Complete the definitions of the constructors, getX and getY. (You'll write the definition of shift in Step 5.)

Step 4: Interactively test your Point class. To do this:

- Right-click on the Point class icon. A pop-up menu will appear. Create a new Point object by selecting the constructor that takes no arguments; that is, new Point(). A Create Object dialogue box will appear. Click OK. A red box with rounded corners will appear in the object bench towards the bottom of the BlueJ window. This box represents the newly created Point object whose name (identity) is point1.

- Right-click on point1 (i.e., right-click on the red box the represents the Point object, not the icon that represents the Point class). A pop-up menu will appear, listing the Point object's methods. Select *Inspect* from the pop-up menu. An Object Inspector window will appear. Use the inspector to verify that the constructor correctly initialized the x and y fields to 0.0.

- Call the getX and getY methods on point1, and verify that both methods return 0.0.

- Which constructor should you use to create a Point object that is initially located anywhere other than (0.0, 0.0)? Create a Point object named point2 that represents a point with coordinates (5.5, 3.5).

- After the icon for point2 appears on the object bench, call the getX and getY methods on point2, and verify that they return 5.5 and 3.5, respectively.

If necessary, correct the class and retest it before moving to Step 5.

Step 5: Read the Javadoc comment for mutator method shift. Complete the definition of the method. Interactively create a new Point object and test this method. Use an object inspector and/or the accessor methods to verify that shift correctly changes the object's state.

Step 6: Enable BlueJ's unit testing tools:

- From the menu bar, select Tools > Preferences... A Preferences dialogue box will appear.

- Click the Interface tab.

- Click the box labelled Show unit testing tools (a check-mark will appear in the box when the tools are enabled).

- Click OK.

Step 7: Class PointTest contains a *suite* of *test cases* that test class Point.

- Compile both classes by clicking the Compile button to the left of the class diagram.
- Right-click on the PointTest class icon.
- Select Test All from the pop-up menu to run all the test cases. A Test Results dialogue box will appear, listing the test cases that were executed. If all the methods you wrote are correct, there should be green check-marks to the left of all the test cases. An x to the left of a test case indicates that it failed.

If any of the test cases fail, you'll need to locate and fix the bugs. Use an object inspector to help you determine where the problems are (e.g., before and after you execute a method, what values are stored in the object's fields? Which values are correct? Which values are incorrect? What section of code in the method you executed changes those values?) Edit the class and rerun the JUnit test cases until every test passes.

Exercise 2 - Developing a Class that Models Line Segments

When working on this exercise, do not change the Point class you completed in Exercise 1. Specifically, do not change the visibility of the x and y fields, do not change the constructor and method signatures (return types, parameter lists), and do not define additional methods in class Point.

Step 1: You have been provided with a very incomplete implementation of class LineSegment. Double-click on the LineSegment class icon. The Java source for the class will appear in an editor window.

Step 2: Read the documentation for class LineSegment. Define two private fields (instance variables) of type Point, named fromPoint and toPoint. **Important: Do not use different names for the fields. Do not define additional fields. The fields' visibility must be private, not public.**

Step 3: Complete the definition of the constructor. Field fromPoint must be initialized with a reference to a new Point object that represents the starting point of the line segment. Field toPoint must be initialized with a reference to a new Point object that represents the ending point of the line segment.

Step 4: Read the Javadoc comments for the getStartingPoint and getEndingPoint accessor methods. Complete the definitions of getStartingPoint and getEndingPoint. (You'll write the definitions of the other methods in subsequent steps.)

Step 5: Interactively test your LineSegment class. To do this:

- Right-click on the LineSegment class icon. Create a new LineSegment object that models a line with starting point (1.0, 3.0) and ending point (4.0, 7.0). Save the object on the object bench.

- Right-click on the red box the represents the LineSegment object (not the icon that represents the LineSegment class). Open an Object Inspector window that displays the fields in the LineSegment object.
- Notice that the "values" stored in fromPoint and toPoint are represented by arrows. This indicates that these fields contain *references* to Point objects.
  - Select the fromPoint field (click on the field to highlight it), then click the Inspect button. A new object inspector will open, showing the Point object referred to by fromPoint. Use the inspector to verify that the Point object referred to by fromPoint represents the point (1.0, 3.0)
  - Select the toPoint field, then click the Inspect button. A new object inspector will open, showing the Point object referred to by toPoint. Use the inspector to verify that the Point object referred to by toPoint represents the point (4.0, 7.0).
- Call the getStartingPoint method. A Method Result dialog box will appear, indicating that the method returned a reference to a Point object. Click the Get button. An icon will appear on the object bench, representing the Point object returned by getStartingPoint. Open an object inspector on the Point object, and verify that it represents the point (1.0, 3.0). (This demonstrates that methods aren't limited to returning values of primitive types; e.g.,int or double. You've now seen that methods can return references to objects.)
- Call the getEndingPoint method. When the Method Results dialog box appears, click the Get button. After the Point object is transferred to the object bench, open an object inspector on the object, and verify that it represents the point (4.0, 7.0).

If necessary, correct the class and retest it before moving to Step 6.

Step 6: Read the Javadoc comment for method length. Complete the definition of the method. (Hint: the line is the hypotenuse of a right triangle.)

Java's Math class provides a method named sqrt that calculates square roots. This method returns a value of type double. To calculate the square root of a real value *x*, call the method this way:

Math.sqrt(*x*).

Create a new LineSegment object that models a line with starting point (1.0, 3.0) and ending point (4.0, 7.0). Interactively call the length method. Does it return the correct value?

Step 7: Read the Javadoc comments for methods moveHorizontal and moveVertical. Complete the definitions of these methods. Interactively test the methods. Note: these methods do not need to create new Point objects. There is a simpler way to move a LineSegment object.

Step 8: Read the Javadoc comments for methods isVertical and isHorizontal. Complete the definitions of these methods.

Recall from your previous programming courses that you should never compare two real numbers p and q for equality this way: $p == q$. Instead, you should compare the absolute value of their difference to a number that is close to 0; for example, $|p - q| < 0.0001$.

Java's Math class provides a method named abs that calculates absolute values. This method returns a value of type double. To calculate the absolute value of a real value x, call the method this way:

Math.abs(*x*)

Test these methods:

- Create a LineSegment object that models a horizontal line. Interactively call the isVertical and isHorizontal methods on the object, and verify that they return the correct values.
- Create a LineSegment object that models a vertical line. Interactively call the isVertical and isHorizontal methods on the object, and verify that they return the correct values.
- Create a LineSegment object that models a line that is neither horizontal or vertical. Interactively call the isVertical and isHorizontal methods on the object, and verify that they return the correct values.

Step 9: Read the Javadoc comments for method slope. Complete the definition of this method. Recall that the slope of a vertical line is undefined. Your slope method should determine if the LineSegment is a vertical line, and in that case, it should return the value Double.POSITIVE_INFINITY. (This is a constant that is defined in Java's Double class.) Interactively test your method.

Step 10: Class LineSegmentTest contains a suite of test cases that test class LineSegment.

- Compile both classes by clicking the Compile button to the left of the class diagram.
- Right-click on the LineSegmentTest class icon.
- Select Test All from the pop-up menu to run all the test cases. A Test Results dialogue box will appear, listing the test cases that were executed. If all the methods you wrote are correct, there should be green check-marks to the left of all the test cases. An x to the left of a test case indicates that it failed.

If any of the test cases fail, you'll need to locate and fix the bugs. Use an object inspector to help you determine where the problems are (e.g., before and after you execute a method, what values are stored in the object's fields? Which values are correct? Which values are incorrect? What section of code in the method you executed changes those values?) Edit the class and rerun the JUnit test cases until every test passes.

Wrap-up

1. With one of the TAs watching, run the JUnit tests for Point and LineSegment. To do this, click the Run Tests button to the left of the class diagram. The TA will note how many test cases pass. The TA will review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.

2. The next thing you'll do is package the project in a *jar* (Java archive) file named lines.jar. To do this:

   2.1. From the menu bar, select Project > Create Jar File... A dialog box will appear. Click the Include source and Include BlueJ project files check boxes. A check-mark should appear in each box. Do not modify the Main class field.

   2.2. Click Continue. A dialog box will appear, asking you to specify the name for the jar file. Type lines or select the BlueJ icon named lines in the list of files. **Do not use any other name for your jar file** (e.g., lab3, my_project, etc.).

   2.3. Click Create. BlueJ will create a file named lines that has extension .jar. (Note: you don't type this extension when you specify the filename in Step 2.2; instead, it's automatically appended when the jar file is created.) The jar file will contain copies of the Java source code and several other files associated with the project. (The original files in your lines folder will not be removed).

3. Before you leave the lab, log in to cuLearn and submit lines.jar. To do this:

   3.1. Click the Submit Lab 3 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag lines.jar to the File submissions box. Do not submit another type of file (e.g., a .java file, a RAR file, a .txt file, etc.)

   3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 3.3.

   3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

      3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

      3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes to the project you're submitting, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".