

## Objectives

After completing the lab you should be able to

1. Implement a class from a specification.
2. Write Javadoc comments for your Java classes.
3. Design and write unit tests for the constructors and methods of a Java class.

**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/>
- *Javadoc Style Guidelines:* <http://java.sun.com/j2se/javadoc/writingdoccomments/#styleguide>
- Chapter 3 of *Big Java*

## Preparation

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

To-Do:	Lab Preparation
<ol style="list-style-type: none"><li>1. Start Eclipse.</li><li>2. Create a new Java project named <code>lab06-xyz01</code>, where <code>xyz01</code> is your login name.</li><li>3. Create the <code>readme.txt</code> file for this lab, insert the banner file with proper updates.</li></ol>	

## Exercise 1

We all have been to sports games where an individual counts the spectators as they enter the arena by using a handheld clicker. In this part of the lab, you will design, document, implement and test a `Clicker` class.

A *public interface* is the list of constructor and method headings that can be used by other programs. Here is the public interface of the `Clicker` class.

<code>public Clicker()</code>	constructs a clicker starting at 0
<code>public void increment()</code>	adds one to the clicker
<code>public void reset()</code>	resets clicker to 0
<code>public int getValue()</code>	returns the clicker value
<code>public String toString()</code>	returns clicker value as a String

Implement a `Clicker` class in Eclipse.

### Instance Fields

Remember that a class description includes three parts — instance fields (also called instance variables), constructors and methods. For the `Clicker` class what instance fields are needed? The key question to ask yourself is “What information must a `Clicker` object maintain or hold?” It in answering this question to look at the public interface: To what information does it provide access? A second key question is “What type is required for each instance field?”

Insert the code for declaring the instance fields needed for the `Clicker` class. Remember that instance fields are always `private` in this course.

### Constructors

Remember that each constructor should initialize all the instance fields.

Work as follows to learn a useful capability of Eclipse: Insert the code for the constructor without typing in any Javadoc comments. Then click somewhere on the constructor’s heading. Select **Source** → **Generate Element Comment** from the menu and a Javadoc comment should appear before the constructor.

This illustrates the recommended way to enter a constructor or method in Eclipse:

- Type the constructor or method heading with an empty block, i.e., the `{}`,
- ask Eclipse to insert the Javadoc comment,
- fill out the Javadoc comment,
- then write the code for the constructor or method.

### Javadoc Comments

In this course from now on, you are **required** for both labs and projects to have an appropriate Javadoc comment before a class heading, before each constructor

heading and before each method heading. A Javadoc comment starts with `/**` and describes the class's, constructor's or method's *purpose* and other information.

For details on Javadoc comments read and study section 3.3 in *Big Java*.

Since the *first sentence* of each Javadoc comment is copied by the Javadoc utility as a summary in the HTML document, it is important that you write the first sentence with care. In this course, you are required to start the first sentence with an uppercase letter and end with a period. It does not have to be a grammatically complete sentence, but it should be meaningful when it is pulled out of the comment and displayed in a summary.

Here is a sample Javadoc class comment for a `BankAccount` class.

```
/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
```

Note that the first sentence may be several lines long assuming there are no blank lines in between. The javadoc utility will extract the summary until the first period. Therefore, be careful in using abbreviations that end with a period.

Since a method implements an operation, the first sentence in the method's Javadoc comment usually starts with a verb phrase.

Note that in this course as per Sun's guidelines, we require third person (descriptive) and not second person (prescriptive) verbs in Javadoc comments. For example, we avoid

```
/**
 * Return the balance. [2nd person. Avoid!]
 */
public double getBalance {
    return balance;
}
```

and require the following

```
/**
 * Returns the balance. [3rd person. Required!]
 */
public double getBalance {
    return balance;
}
```

Spend a few minutes and study the `Point.java` file from last week's lab. Note that its Javadoc comments obey the following rules:

1. The first sentence of each starts with an upper case letter and ends with a period.

2. Each method's sentence starts with a verb phrase.
3. Each verb phrase uses a third person (descriptive) verb, e.g., "Returns."

Read the section "Add description beyond the API name" in the *Javadoc Style Guidelines*: (<http://java.sun.com/j2se/javadoc/writingdoccomments/#styleguide>) about writing comments that provide information beyond the method name.

### Viewing the Formatted Javadoc Comments

Javadoc comments are written in HTML (HyperText Markup Language), the notation used on web pages. For example, to bold a word or phrase you would surround it with `<b>` and `</b>` HTML tags. See Appendix H in *Big Java* for a summary of some HTML tags.

Eclipse has a feature that will show how the Javadoc will appear in the HTML document. Click in the middle of the Javadoc comment of the `setXCoord` method of the `Point` class. Now click on the tab labeled "Javadoc" near the bottom next to the Console tab where the output of a program is shown. You will see the Javadoc comment formatted as an HTML document. As you edit the Javadoc comment, the formatted text in the Javadoc-tabbed panel will change. Add `<b>` and `</b>` HTML tags around the word "coordinate" and you will see it change to bold.

### Running the Javadoc Utility

You can ask Eclipse to run the Javadoc utility for a Project. To try it out, click on the Java Project `lab05-xyz01` of last week's lab. Select **Project** → **Generate Javadoc...** and Eclipse will run the Javadoc utility and place a new folder `doc` in your `lab05-xyz01` project. Click on `Point.java` to select it then select **Navigate** → **Open External Javadoc**. Your default web browser will open and display the HTML-based documentation for the `Point` class.

Spend a few minutes comparing the Javadoc comments in the `Point` class and the HTML document. The form of the HTML document should be familiar to you since Java developers use the same Javadoc utility and the same wording conventions to generate the API web pages.

### Add Methods to the Clicker Class

Insert the code for the four `Clicker` class methods.

In the `toString` method you need to return a `String` value but your `Clicker` holds an `int`. An easy way to do this conversion is as follows:

```
return "" + count;
```

The two double quotes without a space in between specify an empty `String`. Since the empty `String` is first in the expression, the “+” symbol will be interpreted as concatenation operator and the Java compiler will convert the `int` to a `String`.

Remember to fill out the Javadoc comments in a meaningful way obeying the rules described above. Use the `Point` class as an example. Note that Eclipse inserts `@return` in the Javadoc comment for `getValue` method. The “at” symbol (`@`) specifies a tag that is used by the Javadoc utility. You have already seen a tag with your login name in `@author`. You should write meaningful phrases for all `@` tags. That includes changing your login name to your real name.

When you ask Eclipse to generate a comment for `toString`, it inserts the following non-Javadoc comment:

```
/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
```

If you hover the mouse pointer over the method name `toString`, you will see a message in a yellow box that explains what the method should do. Read the contents of the box carefully. Since this method is an *override* of another method (that’s what the green triangle in the margin means), Eclipse is referring you to the original method’s comment. It is OK to leave the comment as it is.

### Test the Class in the `main` Method

In the `main` method insert code to test your `Clicker` class. You are required to follow these guidelines:

1. Test the constructor and each method in the class.
2. The output should describe what you are testing.
3. The output should include the string “Expected:” and expected value.

Use the `main` method of the `Point` class of lab 5 as a model of the form to be used.

In your `readme.txt` file insert the output of a run of your finished `Clicker` class.

## Exercise 2

Create an `Attendance` class with a `main` method that constructs four `Clicker` objects, one for each of the four doors of a sports arena. In the North door 3 people enter, in the South door 2 people enter, in the West door 5 people enter and no one enters the East door. Have the program print the total number of people who attend the event. Remember to have appropriate Javadoc comments.

In your `readme.txt` file insert the output of the `main` method of your finished `Attendance` class.

## Exercise 3

An employee has a name (a string) and a salary (a double). The public interface includes a constructor with two parameters

```
public Employee(String employeeName, double currentSalary)
```

and methods

```
public String getName()
public double getSalary()
public void raiseSalary(double byPercent)
public String toString()
```

These methods return the name and salary, raise the employee's salary by a certain percentage and return a representation of an `Employee` object as a `String`.

### Details

Write appropriate Javadoc comments and a `main` method that tests the `Employee` class.

Construct two employees: Harry Potter with starting salary of \$1000.00 and Ron Weasley with starting salary of \$200.00. After salary increases of 10% and 5% respectively, print their names and salaries.

In your `readme.txt` file insert the output of a run of your finished `Employee` class.

## What to Submit

- Your `readme.txt` file should include the results from the three exercises.
- 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](#)