CSCI 203: Introduction to Computer Science I
Bucknell University
Computer Science Department
http://www.eg.bucknell.edu/~csci203

Lab 3: Objects, Methods, and Errors

Table of Contents

1. Objectives

After completing the lab you should be able to

1. Learn how to use Eclipse.

2. Identify and fix simple coding errors in Java.

3. Know the basics about Java and programming errors, including more advanced syntax errors, semantic errors, and intent errors.

4. Become more comfortable asking questions.

References: You will use the following web pages in this lab.

- *CSCI 203 course website*
- *Java API website*
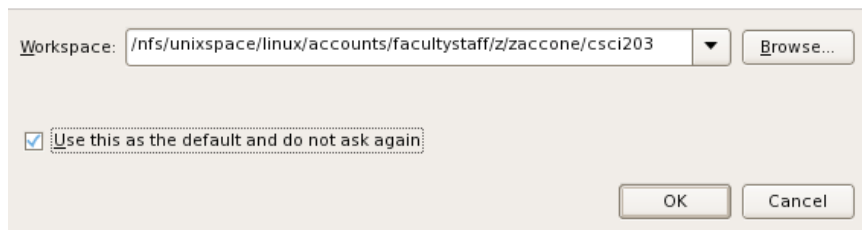
2. Getting Started With Eclipse

Start Eclipse by either typing

```
eclipse &
```

in your terminal window, or by selecting Eclipse from the Programming menu found within the Red Hat menu. Eclipse will present you with a dialog asking you for the location of your workspace. Indicate that it is your `csci203` directory and check the box so it won't ask again.

Select a workspace

Eclipse SDK stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /nfs/unixspace/linux/accounts/facultystaff/z/zaccone/csci203 ▼ | Browse...

☑ Use this as the default and do not ask again

OK | Cancel

When you start Eclipse you will be presented with the following welcome screen.
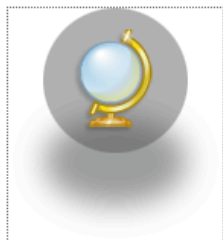
Click on the **Workbench** icon to begin using Eclipse.

2.1. Change Eclipse Settings

Follow the instructions in this section to make sure Eclipse is configured properly.

Open Eclipse preferences by selecting **Window** → **Preferences**.

- In **Java** → **Editor** → **Folding**, uncheck Header Comments and Imports. Enable folding should remain checked. Click the **Apply** button.

- In **Java → Editor → Save Actions**, check the box that says "Perform the selected actions on save". Then check the box that says "Format source code." Click the **Apply** button.

- In **Java → Code Style → Code Templates**, click the disclosure triangle next to Comments. Click on the word Files and then press the **Edit. . .** button. Enter a comment similar to the one that follows, using your name.

```
/**
 * CSCI 203, Rick Zaccone
 * ${date}, ${time}
 */
```

Click **OK**, then click the **Apply** button.

- In **General → Editors → Text Editors**, check the box that says "Show print margin" and then click the **Apply** button (you may need to scroll down to see the button).

- Click the **OK** button to exit the preferences.

3. Practice Entering a Java Program

In this exercise, you are asked to type the source code of a complete program into a file, then run the program.

- Select **File → New → Java Project**.
- Enter lab03-xyz001 as the project name where xyz001 is your login name. Press the **Finish** button.

- In the Package Explorer pane on the left, click on the disclosure triangle next to your `lab03-xyz001` folder. Click once on the `src` folder so it is highlighted.
- Select **File → New → Class**.
- Enter `AreaTester` as the name.
- Check the box that says to create a method stub for `public static void main`.
- Check the box that says to "Generate comments".
- Click the **Finish** button.

You should be looking at a new class named `AreaTester` which has a `main` method. There are comments too!

Finish entering the program so that it looks like the one that follows. The only differences will be the name, time and date. Save your work periodically.

```java
import java.awt.Rectangle;

/**
 * CSCI 203, Rick Zaccone
 * Sep 13, 2010, 5:04:47 PM
 */

/**
 * Construct a rectangle with area 51.
 *
 * @author zaccone
 */
```

```java
public class AreaTester {

    /**
     * @param args
     *                 Unused
     */
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(3, 4, 6, 7);
        double area = rect.getWidth() * rect.getHeight();
        System.out.println("Area: " + area);
        System.out.println("Expected: 51.0");
    }
}
```

To run the program, save it, then select **Run** → **Run** or click on the run icon at the top of the window. It's a green circle with a right pointing arrowhead. A console window will appear at the bottom of the Eclipse window.

The program is constructing a `Rectangle` object and then computing the area. The intent of the program is to construct a rectangle with area 51, but the program gives the wrong answer.

Visit the Java API website and find the API for the `Rectangle` class. Find the constructor that this program uses and read how it works. Correct the program so that the area of the rectangle is 51.

4. Perimeter of a Rectangle

Select **File** → **New** → **Class** (or click on the green circle with a white C) to create a new class. Call the class RectanglePerimeter, check the box that asks Eclipse to create a method stub for main, check the box that says "Generate comments" and press the **Finish** button.

   Use the previous program as a guide for this exercise. Complete main so that it constructs a Rectangle object using the same parameters as the solution to the previous section and then computes and prints its perimeter. Use the getWidth and getHeight methods. Also print the expected answer. You can use the window tabs to switch between programs.

5. Copying Files

Throughout the lab, you will be working with some Java programs. You'll read the programs, edit them, fix errors, and run these programs. The first task you should complete here is to copy the files from the course directory to your own lab directory.

   To copy the files, follow these steps.

1. In the Eclipse Package Explorer (the pane on the left), click on the src directory so it is highlighted.

2. Select **File** → **Import...**

3. In the window that appears, click on the disclosure triangle next to **General** and then click on **File System**. Press the **Next** button.

4. At the top of the **Import** window you need to indicate a directory from which you are importing. Click the **Browse...** button.

5. Click on the **accounts** button at the top of the **Import from directory** window. Double click on `COURSES`, then `csci203` (*not* `cs203`), `2010-fall`, `student` and finally `labs`. Click on `lab03` and press the **OK** button.

6. Check the box next to `lab03` and press the **Finish** button.

6. Create a Read Me File

Select **File** → **New** → **Untitled Text File**. Select **File** → **Save As...**, select your lab folder as the parent folder and save the file as `readme.txt`. Open your `banner` file (**File** → **Open File...**) and copy and paste your banner at the top of the file. Modify the date and lab portions and save again.

   Note: Many labs this semester will contain exercises that require you to answer a question in your `readme.txt` file. Before answering the question, place a heading in your `readme.txt` file so it will be easy for the person grading the lab to determine which question you are answering. For example, the heading for the exercises in the **To-Do** box titled *Division by Zero* would look as follows:

```
/*===================================
 * Division by Zero
 *===================================
 */
```

   The sub-heading for the Exercise 3 of Lab X would look as follows:

```
/*===================================
```

```
 * Exercise 3 of Lab X
 *====================================
 */
```

7. Experience with Coding Errors

This exercise gives you more practice with the error messages that a Java compiler might give you. Programs written in a particular programming language such as Java have to follow syntax rules so the compiler will understand what the program means. Do you remember your experiences with the IBM code? If you mean to store the value currently in the accumulator into memory location 1, you'd use the code 101. If by accident you wrote 901 instead of 101, IBM won't recognize this instruction. This type of error is called a *syntax error* when the program is written incorrectly. On the other hand, if you wrote the code as 102 instead of 101, IBM will recognize this as syntactically correct code. It means store the result in the accumulator into memory location 2. But this was not your intention. This type of errors is called a *semantic error*. This segment of the exercises will give you practice recognizing and dealing with different types of errors in Java.

To-Do:	Experience with Various Types of Errors
• Complete the three tasks listed below and record your responses in your `readme.txt` file.	

Below is a series of steps to be performed along with some questions that you need to answer. Type the answers into your `readme.txt` file.

1. Open the file `ErrorTest1.java` by double clicking on it in the Package Explorer. There is an indicator in the left margin that shows an error. Hold the mouse over the indicator and Eclipse will tell you what's wrong. Explain what the error message means.

2. Now change the declaration of `z` so that it reads

   ```
   double z = 0;
   ```

   Save and run the program.

3. Modify the corrected program according to the instructions below. For each modification, examine the error message you get. Provide a brief explanation of *both* what is wrong with the program, *and* how the Eclipse reports the error.

   Note: Since this program is short, it will be easy to make changes based on line number by just counting the lines. With larger files, this is more difficult. To move to a specific line in a file within Eclipse, select **Navigate → Go to Line…** Eclipse shows you the current line number at the bottom of the window. If the display shows 11:27 (for example) it means that you are on line 11, position 27.

   Note: You should undo (**Edit → Undo Typing**) each change listed below before moving on to the next part. As a precaution, you should save your `readme.txt` file after explaining each error. Yet again we note that it is a good habit to save your edit sessions frequently.

   (a) Put a comment on line 3 by adding `//` in front of `{`.
   (b) Put a comment on line 4 by adding `//` in front of `{`.

(c) Comment line 6 by adding `//` in front of `double`.

(d) Remove the closing double quote mark from line 9.

(e) Replace (`String[] args`) after the word `main` with a semicolon (`;`) in line 4.

(f) Change the word `double` in line 6 to `int`.

(g) Change the word `double` in line 6 to `integer`.

Make sure to put the responses to these seven tasks into your `readme.txt` file.

8. Dealing with More Errors

In previous section, you dealt with some Java syntax errors. The exercises in this section will show you a variety of programming errors one may encounter.

To-Do: **Running** `ErrorTest2.java`

1. Close `ErrorTest1.java` and open `ErrorTest2.java`.

2. Run the program.

3. Read the program and understand why it gave you the output you see.

When you run the Java program, you should see the program results as follows:

```
CSCI 203 is fun!
CSCI 203 is fun!
a is 3
b is 16.0
```

Make sure you understand which lines of the program caused what output. Why was CSCI 203 is fun! displayed twice? You don't have to write any responses for this, just think it through.

8.1. Syntax Errors

As discussed in the previous section, syntax errors are the mistakes the programmer makes when the source code does not follow the Java syntax rules. These errors are detected when the program is compiled, that is, when the program is translated from source code to Java byte-code. The following **To-Do** will ask you to purposely create some syntax errors, observe any error messages, and then fix the errors.

To-Do: **Syntax Errors**

1. Remove the double-quote after `CSCI 203` on line 8 and save the file. Note the error. Fix the error

2. Remove the *semicolon* from the end of line 22. Note the error. Fix the error.

Study the error messages. It is important that you learn how to deal with error messages. The first error message said:

`String literal is not properly closed by a double-quote`

This is how Eclipse typically reports errors. It tries its best to point out where the error occurred. You can't run the program until you have fixed all of the errors.

In the above description, we said Eclipse will *try its best* to locate where the error is. Sometimes this is not an easy task. In the following **To-Do** box, you'll find that the error messages sometimes aren't clear. Syntax rules alone are often not enough to determine what a programmer intended. For example, consider the following Java statement:

```
int number = "1234";
```

One cannot tell if the programmer mistakenly declared the variable `number` as an integer when it should be a `String`, or if the value assigned to the variable should have been in integer instead of a string.

Now try the following.

To-Do: **Inaccurate Reporting of Errors**

1. Remove the double-quote before `is` on line 9 in `ErrorTest2.java`.

2. Read the error that Eclipse reports.

3. After you are sure you understand what's going on, add the double-quote back in so that the program is correct.

4. Save the file.

8.2. Semantic Errors

Semantic errors are errors in meaning. When there are semantic errors, the program will produce wrong results even though the syntax is correct. For example, if an operation is meant to be division but the programmer put in an addition operator instead, the results will be wrong.

To-Do: **Semantic Errors**

1. In `ErrorTest2.java`, comment out line 29 and add a new assignment statement before it as follows:

   ```
   double b = Math.sqrt(-2.0);
   // double b = Math.pow(2.0, 4.0);
   ```

2. Run the program.

3. Copy the error message error to your `readme.txt` file.

We know something funny should happen when we run the program since we shouldn't be able to take the square root of a negative number! The error which results is a *semantic error* — an error due to wrong meaning — and is signaled by `NaN` (Not a Number) being reported as the value of `b`. As far as Java is concerned, the syntax of the program is still correct. This is one way a running program can report a faulty computation.

Another related problem involves division by zero. Mathematically the result of dividing by zero is undefined. However, if a programmer doesn't design a program carefully, sometimes the execution of that program will result in a division by zero. When a division by zero occurs, Java will report the problem. The following **To-Do** exercise will test this case:

To-Do: **Division by Zero**

1. In `ErrorTest2.java` replace the expression `Math.sqrt(-2.0)` by the expression `a/0`, that is, integer `a` divided by zero.

2. Run the resulting program. The following error message will be printed:

   ```
   Exception in thread "main" java.lang.ArithmeticException: / by zero
   at ErrorTest2.main(ErrorTest2.java:29)
   ```

Leave the program `ErrorTest2.java` as it is for now.

8.3. More Semantic Errors

Some semantic errors are caught by the compiler. For example, if the argument (operand) passed to the square root method is a string instead of a number, the operation would make no sense. Another simple example of this type would be to apply arithmetic operations such as division to a string. The compiler can catch this type of error. The following **To-Do** illustrates this idea.

> **To-Do:** **More Semantic Errors**
>
> 1. In ErrorTest2.java, replace a/0 with Math.sqrt(s3).
>
> 2. Study the error message. In this case, Eclipse can detect the improper use or meaning — you can't take the square root of a string object.
>
> 3. Before continuing, change the argument of Math.sqrt from s3 to 2.0 (not -2.0) so the program now is correct.
>
> 4. Run the program to see the results.

8.4. Java Strings

Because Java String is a very powerful class that can be used in many different applications, and because we touched upon Java Strings in the above example, please read the Java API (Application Programming Interface) for the String class and become familiar with Java strings.

The Java API is well documented at its website. The CSCI 203 course website has a direct link to the Java API.

> **To-Do:**   **Reading the Java** `String` **API**
>
> 1. Go to the Java API website.
>
> 2. Look at the Java API and find the `String` class.
>
> 3. Browse through the `String` class description to become familiar with the kind of information provided by the Java API.
>
> 4. In your `readme.txt` file list the first five methods (not constructors) provided for the Java `String` class.

8.5. Intent Errors

A program may run to completion without being stopped by errors, but it may not be correct. The result of the computation may not be what the programmer intended. In such cases we say that an *intent error* has occurred. Intent errors in a broad sense are also semantic errors. For example, when working with a circle object, a programmer wanted to print the value of the radius. Instead the value of the diameter is printed. The Java compiler (or a compiler of any other language) has no way to detect this type of error. Do the following exercise to appreciate this type of error:

To-Do: **Intent Errors**

1. In `ErrorTest2.java`, modify the final System.*out*.println statement to the following:

   ```
   System.out.println("b is " + a);
   ```

2. Run the program.

3. In your `readme.txt` file, write your thoughts on why the output is not correct.

---

9. What To Submit

Make sure that your `readme.txt` file contains the following.

1. Answers to the seven questions following the **To-Do** box *Experience with Various Types of Errors*.

2. Answers to the questions in the **To-Do** box *Semantic Errors*.

3. Answers to the questions in the **To-Do** box *Reading the Java String API*.

4. Answers to the questions in the **To-Do** box *Intent Errors*.

Open a File Browser by double clicking on the home folder icon on your desktop. Double click on your `csci203` folder. Drag your lab folder onto the CSCI203 *lab* drop box on the left:

CSCI203 Lab drop_box