

CSCI 203: Introduction to Computer Science I
Bucknell University
Computer Science Department

<http://www.eg.bucknell.edu/~csci203>

Lab 8: Decisions

Table of Contents

1. Objectives
2. Preparation
3. Introduction
4. Comparing Double Values
 - 4.1. Exercise 1: Does Comparing Doubles Obey Math Laws?
 - 4.2. Exercise 2: Writing an `areClose` Method
5. Comparing `String` Objects
 - 5.1. Exercise 3: Comparing `String` Objects
6. Comparing Objects
 - 6.1. Exercise 4: Comparing `Point` Objects
7. What to Submit

1. Objectives

After completing the lab you should be able to

1. Write `if` statements.
2. Compare `double` values that are close.
3. Work with the `equals` and `compareTo` methods for the `String` class.
4. Compare objects for equality and test for `null`.

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

- *CSCI 203 course website*
- *Java API website*
- Chapter 5 of Big Java

2. Preparation

Prepare for the lab by completing the following steps.

1. Start Eclipse.
2. Create a new Java project named `lab08-xyz01`, where `xyz01` is your login name.
3. Create the `readme.txt` file for this lab, insert the banner file with proper updates.

3. Introduction

In today's lab we explore different ways to make decisions in a program. The primary way is to use `if` statements that select two different groups of statements depending on a condition. In Java, the condition is a logical expression (also called a Boolean expression) that evaluates to either true or false.

```
if (logical expression) {  
    // statements to do if logical expression is true  
} else {  
    // statements to do if logical expression is false  
}
```

The `else` part is optional.

Comparing `int` values is rather easy. For example,

```
int a = 3;  
int b = 4;  
  
if (b == a) {  
    System.out.println("b is equal to a");  
} else {  
    System.out.println("b is not equal to a");  
}
```

However, comparing `double` values and String objects can be tricky.

4. Comparing Double Values

4.1. Exercise 1: Does Comparing Doubles Obey Math Laws?

Create a new `CompareDoubles` class and enter the following code in the main method.

```
double d = 4.35;
double f = d - 0.2;
if ((d - f) == 0.2) {
    System.out.println("Comparing doubles obeys Math laws!");
} else {
    System.out.println("Comparing doubles does not obey Math laws!");
}
System.out.println("d - f: " + (d - f));
System.out.println("Expected: 0.2");
```

In your `readme.tex` file, describe what happened and why.

4.2. Exercise 2: Writing an `areClose` Method

If you need to compare `double` values for equality or inequality, you should always check to see if the values are within a small tolerance (traditionally called `epsilon`). The idea is to subtract the two values and see if the result is less than or equal to `epsilon`. We need to take the absolute value because the result may be negative. See pages 188–189 in *Big Java*.

To your `CompareDoubles` class add a `static` method with the following method header:

```
public static boolean areClose(double x, double y, double epsilon)
```

Since this method returns a `boolean` value, i.e., either true or false, it is called a *predicate* method (see page 205 in *Big Java*). It is convention to name predicate methods starting with a form of the verb “to be” as in `isLetter`.

You are to implement the method such that it returns true if the values `x` and `y` are close, i.e., within `epsilon` and false otherwise.

Add to your main a check that uses your `areClose` method to see if $(d - f)$ is close to 0.2 within an `epsilon` of 10^{-14} .

Place a copy of your `areClose` method, your main method and a run in your `readme.tex` file.

5. Comparing String Objects

Two objects of a primitive type such as `int` can be compared by using the comparison operators `<`, `==`, or `>=`. These operators should not be used for comparing non-primitive objects, however. Some classes do provide methods that facilitate such comparisons. In this section we will focus on the `String` class to understand how to compare several strings.

Strings have a natural ordering: alphabetic ordering. Strings in Java, however, can have nonalphabetic characters as well as upper- and lowercase letters. Thus we need a generalization of alphabetic order to compare Java Strings: *lexicographic order*. Lexicographic order is similar to alphabetic order, with the addition that when we need to compare two characters, we use the values of their *Unicode* representation which is a standard to encode languages across the world into a computer recognizable form (See table on pages

1058–1059 in *Big Java*.)¹. For example, uppercase letters come before lowercase letters in lexicographic order.

The `String` class has four methods that can be used to compare two string objects: `equals`, `equalsIgnoreCase`, `compareTo`, and `compareToIgnoreCase`. As you can probably guess, these methods are in two pairs, where comparison is done either taking into account the case of letters or ignoring it — otherwise the methods behave the same way. The method `equals` returns a boolean value based on whether the two strings are the same. The expression

```
"this".equals("This")
```

will return `false`, but if the `equalsIgnoreCase` method had been called then the expression would evaluate to `true`.

The use of the two `compareTo` methods is more complex. These methods return an integer value that tells whether one value comes before, at the same point, or after the other in lexicographical ordering (in Unicode). If `str1` and `str2` are `String` objects, the message

```
str1.compareTo(str2)
```

will return a value as described below.

¹We formerly used ASCII (American Standard Code for Information Interchange) encodings when English was about the only language used in computing. Now with many other languages used in computing, ASCII is insufficient. Unicode was introduced to solve this problem. The original ASCII code is now a subset of Unicode.

- 0 if the contents of `str1` and `str2` are the same.
- an integer less than 0 if `str1` would appear before `str2` in a lexicographic ordering;
or
- an integer greater than 0 if the reverse of the above is true.

The results of the method `compareToIgnoreCase` would be as suggested by the name of the method.

5.1. Exercise 3: Comparing String Objects

Create a new `CompareStrings` class. Add your code to the main method.

For each of the following comparisons, state whether the returned value is negative, zero, or positive if the method returns an integer; `true` or `false` if the method returns a `boolean`.

Type your answers into your `readme.txt` file.

1. `"apple".equals("book")`
2. `"coma".equals("comma")`
3. `"soup".equalsIgnoreCase("SoUp")`
4. `"apple".compareTo("book")`
5. `"comment".compareTo("comma")`
6. `"Easy".compareTo("Easy")`
7. `"easy".compareTo("Easy")`

8. `"easy".compareToIgnoreCase("Easy")`
9. `"Horse".compareTo("buggy")`
10. `"needles".compareTo("needless")`
11. `"then".compareTo("the")`

See table of ASCII values on pages 1058–1059 in *Big Java*.

6. Comparing Objects

Import the `Point.java` file from `~csci203/2009-fall/student/labs/lab08`. This version of the `Point` class is close to the previous one you have seen except the instance fields are `int`, it now has an `equals` method and several new tests of the method. Open the program and study the `equals` method. Note that it returns a `boolean` type, i.e., a value that is either true or false. Run the program to see the results.

If you want to compare two objects of a class you write, you need to write your own `equals` method. The default `equals` method will only check to see if the objects are identical, i.e., the references (think addresses) are the same.

6.1. Exercise 4: Comparing `Point` Objects

At the end of the `main` method construct two more points, say `p3` and `p4`, with values both at `(1,3)`. For each of the following five parts you don't need to print the expected value just use an `if/else` that prints the result of the test.

1. Write an `if` statement that compares the two `Point` objects using the `==` operator. Copy the results to your `readme.txt` file and explain your result.

2. Write a second `if` statement that compares the two `Point` objects using the `equals` method. Copy the results to your `readme.txt` file and explain your result.
3. Now assign `p4` to `p3` and follow it with copy of the above two `if` statements. Copy the results to your `readme.txt` file and explain your result.
4. Now assign `null` to `p4` and follow it with an `if` statement to test if `p4` is `null`. Do you use the `==` operator or the `equals` method? Answer the question in your `readme.txt` file and explain your result.
5. What happens if you now send `p4` the message `getXCoord`? Copy the results to your `readme.txt` file and explain your result.

7. What to Submit

Your `readme.txt` file should include the results from the four exercises.

Make sure your `readme.txt` file is in your lab directory, and drag your lab directory to the CSCI 203 lab drop box: [CSCI203 Lab drop_box](#)