Lab 12: More on equals, compareTo and Arrays

Let's get started!

Today we'd like you to:

1. **Open Eclipse**. Remember to keep all your projects in the same workspace. Think of a workspace as a shelf, and each project is a binder of stuff on the shelf. If you need some help organizing your workspace, we'd be happy to help!
2. **Create a new Java project**. Call it Comp1510Lab12*LastNameFirstInitial*.
3. **Complete the following tasks**. Remember you can right-click a Java project's src package in the Eclipse Package Explorer pane (Window > Show View > Package Explorer) to quickly create a new Java class, enum, interface, etc.
4. When you have completed the exercises, **show them to your lab instructor**. Be prepared to answer some questions about your code and the choices you made.

What will you DO in this lab?

In this lab, you will:
1. review key concepts of good class design
2. apply a standard algorithm for overriding the equals method
3. explore how to add and store matrices using 2D arrays

Table of Contents

1. Providing a proper equals method

This term we have explored the concept of a class, and we have learned some good rules about class design. For example:

1. We like our classes to **encapsulate** data and the methods that work on that data.
2. Each class represents a **single coherent concept**.
3. Each class contains private **instance variables that store the state of an object** or instance of the class. We know that sometimes these variables cannot change (they're not variable at all!) and we say things like the variables are final, or constant, or immutable.
4. We know that a class has at least one constructor which is used to create objects, or instances of the class. The **constructor is responsible for initializing the instance variables**, i.e., the constructor generates the starting 'state' of the freshly made object, often using values passed as arguments from a client object or driver.
5. Each class has a **toString** method which returns a String representation of an object's state.
6. Each class has **accessors** for some, possibly all of its instance variables.
7. Each class may have some **mutators** for some of its instance variables (but we generally like to minimize the mutability of a class as much as possible because fewer moving parts means fewer problems!).
8. For objects of a class to be comparable to one another, the class should implement the **Comparable** interface, which forces the class to provide a compareTo method. Remember that we are actually implementing the interface Comparable<*theClass*>, where *theClass* is the name of the class, i.e., String implements Comparable<String>, and Integer implements Comparable<Integer>.

Classes should also provide a proper equals method. Next term you will see that the equals method is inherited from the Object class, the class at the top of the Java inheritance hierarchy.

(Classes also inherit the toString() method from the Object class. Have you taken a peek at the source code for the Object class, or the String class? When you downloaded the JDK, one of the files you downloaded was called src.zip. Find it and unzip it. Behold! The source code for the entire Java library!)

Let's create an immutable Player class and make sure it provides a few important methods that are defined in the Object class (but need to be redefined to be appropriate for our class).

1. **Create a Java class called Player**. The class should contain the following private instance variables:
     i. String to store the name
     ii. String to store the team name
     iii. An int to store the player's jersey number.
2. The Player class needs a **constructor**. It accepts three parameters, one for each instance variable. The constructor must enforce the following:
     i. A player's name cannot be null or a String composed of whitespace. If the parameter passed to the constructor is null or a String composed of whitespace, the constructor must throw an IllegalArgumentException with a helpful, meaningful message.

ii.   A player's team cannot be null or a String composed of whitespace. If the parameter passed to the constructor is null or a String composed of whitespace, the constructor must throw an IllegalArgumentException with a helpful, meaningful message.

iii.   A player's jersey number cannot be zero or negative. If the parameter passed to the constructor is zero or negative, the constructor must throw an IllegalArgumentException with a helpful, meaningful message.

3. Remember to **document** the rules for what is allowed and not allowed when creating a new Player in the constructor's Javadoc. This is so developers reading your class's API know how to use your Player correctly.

4. Your Player class is **immutable**. What does that mean about the instance variables?

5. Players should be easy to compare. **Modify your Player class so it implements the Comparable<Player> interface**. You will have to implement the compareTo method. Sort Players by jersey number in ascending order. That is, when comparing two Players, the player with the lowest jersey number is first, and the player with the higher jersey number is second.

6. Okay let's redefine our first Object method. Implement a **toString** for the Player. The String returned by our toString method should contain the name, the team, and the jersey number. Label the data so the user can read it easily.

7. Let's redefine our second Object method: implement an equals method for the Player. The equals method should accept a single parameter of type Object called other, and return a boolean result indicating whether the parameter is "equal to" this one (the Player executing the method). Here's the algorithm you should use inside the method:

   a. Check if the parameter equals null. If it does, then the Player executing the equals method is obviously not equal to it, so return false.

   b. Check if the Player executing the method ("this") and the parameter are aliases. If so, then they are obviously two references to the same object, and we can return true. (Hint: remember we can compare two objects to see if they are aliases using ==.)

   c. Check if the Object passed as a parameter is an instance of the Player class. If it's not, then it's obviously not equal to "this" Player executing the code and we can return false. We can check if the Object is a Player using the getClass method, which is also (surprise!) inherited from Object. Everything in Java has a getClass method. We can use it like this:

```
1. if (!getClass().equals(other.getClass())) {
2. return false;
3. }
```

   d. Finally, if the parameter is not null, and it's not an alias, and it IS a Player, we can cast it as a player, and compare its name, team, and jersey number to "this" Player's name, team, and jersey number. Return true if the names, teams, and jersey numbers are the same, otherwise return false.

8. Eclipse may complain after you create your equals method. It might say you need a hashCode method. We will learn about hashing and hash codes in COMP 2526. You can ignore this warning. The consequences of ignoring the warning means our Player class will not work properly with some of the container classes that we have not seen in this course.

9. Finally, let's create a driver class called **ComparePlayers**. ComparePlayers should:
    a. Contain a **main** method that drives the program.
    b. Create an **array of Player** large enough for 2 Players.
    c. **Use a try-catch to try to create a Player with a null name or a name with only whitespace** (just hardcode the parameters passed to the Player constructor), and catch the IllegalArgumentException and print its message
    d. **Use a try-catch to try to create a Player with a null team or a team with only whitespace** (just hardcode the parameters passed to the Player constructor), and catch the IllegalArgumentException and print its message
    e. **Use a try-catch to try to create a Player with a negative jersey number** (just hardcode the parameters passed to the Player constructor), and catch the IllegalArgumentException and print its message
    f. Ask the user for the name, team, and jersey number for a **Player**, instantiate a new Player, and store it in the array.
    g. Ask the user for the name, team, and jersey number for a **second Player**, instantiate a new Player, and store it in the array. (Hint: reduce duplicated code by using a loop to do this and step f!)
    h. Report the result when the first Player is compared to the second Player using the **compareTo** method.
    i. Report whether the two players are **equal**.

2. Matrices (aka 2D arrays)

Let's create a Matrix class. Recall from COMP 1113 that a matrix stores a 2D array of numbers. We can add matrices if they are the same size, and we can multiply matrices if they are the correct size. In this lab, let's only consider adding and leave other operations for future study.

We will assume that all users of this class count in the Java style, i.e., rows and columns begin with row 0 and column 0.

1. Create a class called **Matrix**.
    a. A Matrix has a single private instance variable which is a **2D array of int**.
    b. A Matrix needs a **constructor**. The constructor accepts two integers for the number of rows, and the number of columns (in that order!) and initializes the 2D array of int to be that size.
    c. Implement a method called **getRows** which returns the number of rows in the Matrix.
    d. Implement a method called **getColumns** which returns the number of columns in the Matrix.
    e. Implement a method called **getValue** which accepts two integers for row and column respectively and returns the int stored at that location in the matrix.
    f. Implement a method called **setValue** which accepts a int for a new value, an int for the row, and an int for the column. The new value should be put into the matrix at the specified row and column.

4 of 6

g. Implement a **toString** method which should print the Matrix in matrix form. You will probably need to use a nested for loop to build and then return a concatenated String.

h. Implement a method called **add** which accepts a Matrix and returns a Matrix. This method should create a new Matrix which contains the sum of the contents of "this" Matrix and the Matrix passed as a parameter. Return the new sum Matrix. If the Matrix object passed as a parameter is the wrong size, this method should throw a new IllegalArgumentException.

i. Implement a method called multiply which accepts a Matrix and returns a Matrix. This method should create and return a new Matrix which is the product of this and the parameter, according to the rules you learned in your Math class. If the Matrix object passed as a parameter is the wrong size, this method should throw a new IllegalArgumentException (recall that the number of columns of the first matrix has to equal the number of rows of the second matrix).

2. Every class needs some good tests. Create a driver file called MatrixDriver which contains a main method.

a. In the main method, instantiate a square matrix that stores these values: $\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$.

b. Instantiate a second square matrix that stores these values: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$.

c. Print the toString for both matrices. Does the toString position the numbers correctly?

d. Add the two matrices and assign the result to a third matrix variable. Print out the sum matrix. Is it correct?

e. Create a rectangular matrix that looks like this: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$. Try adding it to one of the square matrices. Can you catch the exception and print the error for the user?

f. Multiply the matrix in b) times the matrix in e). The result should be a 2 x 3 matrix. Did you get the right answer? Multiply the matrices in the reverse order. You should throw an exception.

3. You're done! Optionally submit your work.

If your instructor wants you to submit your work in the learning hub, export it into a Zip file in the following manner:

1. Right click the project in the Package Explorer window and select export…
2. In the export window that opens, under the General Folder, select Archive File and click Next
3. In the next window, your project should be selected. If not click it.
4. Click *Browse* after the "to archive file" box, enter in the name of a zip file (the same as your project name above with a zip extension, such as Comp1510Lab12BloggsF.zip *if* your name is Fred Bloggs) and select a folder to save it. Save should take you back to the archive file wizard with the full path to the save file filled in. Then click Finish to actually save it.

5. Submit the resulting export file *and* your answers document as the instructor tells you.