Lab 10: Arrays and Inheritance

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 something like COMP 1510 Lab 11.
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.
5. **Remember that for full marks your code must be properly indented, fully commented, and free of Checkstyle complaints**. Remember to activate Checkstyle by right-clicking your project in the Package Explorer pane and selecting Checkstyle > Activate Checkstyle.

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. create a simple inheritance hierarchy that explores abstract classes and overriding methods
4. explore how to add and store matrices using 2D arrays
5. incorporate inheritance and interfaces in a feathery inheritance hierarchy.

Table of Contents

1. Overriding the 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 an overridden **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 Comparable<Class>, where Class is the name of the class, i.e., String implements Comparable<String>, and Integer implements Comparable<Integer>.

Classes should also override the equals method. 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 yet 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 overrides a few important methods from the Object 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 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 override 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 override 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 if 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.
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 c!)
    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. A gentle introduction to inheritance

Inheritance is one of the hallmarks of object oriented programming. Although we are teaching you about inheritance using Java, the concepts can be applied to many popular programming languages like Python and C#. The ability to extend classes promotes reusability and makes our code easier to understand and maintain. It also supports polymorphism which we will introduce next week. For now, let's experiment with a simple inheritance hierarchy:

1. Create an **abstract Dog** class. Remember abstract classes have the word abstract in the class header. An abstract class cannot be instantiated (like an interface!).
    a. A Dog has a **name**. Use the protected visibility modifier instead of private.
    b. The **constructor** should accept a String for the name, and throw a NullPointerException if the parameter is null or composed entirely of whitespace.
    c. Generate a public **accessor** for the name. Do not create a mutator. The name is immutable, which makes our Dog class immutable.
    d. Create a public method called **speak** which returns a String that says WOOF!
    e. Create an **abstract** method called averageBreedWeightKG which accepts no parameters and returns a double. Remember an abstract method has no implementation, it is just a method header followed by a semi-colon.

   f. Generate a **toString** for the Dog class and an **equals** method. Dogs are equal if their names are equal.
2. Create a class called **Labrador**. Labrador must extend the Dog class. A Labrador is a non-abstract subclass of Dog, In the Labrador class header, write extends Dog.
   a. When we **extend Dog**, we must implement the Dog's abstract method(s). If Labrador does not implement the Dog's abstract method(s), it must also be abstract.
   b. A Labrador has a **color** (String) which is private. A Labrador also contains an integer constant called LABRADOR_WEIGHT which stores 35.
   c. A Labrador's **constructor** accepts a String for the name and a String for the color. Pass the name to the superclass (Dog) constructor using super. Make sure the color is not null or composed entirely of whitespace. If it is, throw an IllegalArgumentException, otherwise store the color in the Labrador's color instance variable.
   d. Override the **speak** method. A Labrador says BOW WOW!
   e. The overridden **averageBreedWeightKG** method should return LABRADOR_WEIGHT.
   f. Override the **toString** and the **equals** methods. The toString should contain the name and the color. Labradors are equal if their names are equal AND their colors are equal.
3. Create a class called **Yorkshire**. Yorkshire must also extend the Dog class. A Yorkshire is a non-abstract subclass of Dog, In the Yorkshire class header, write extends Dog.
   a. When we **extend Dog**, we must implement the Dog's abstract method(s). If Yorkshire does not implement the Dog's abstract method(s), it must also be abstract.
   b. Did you know Yorkshires were originally bred to catch rats in English factories? The Yorkshire stores a boolean called ratter which is true if the Yorkshire is trained to catch rats, and falso if it is just a lap dog. A Yorkshire also contains an integer constant called YORKSHIRE_WEIGHT which stores 4.
   c. A Yorkshire's **constructor** accepts a String for the name and a boolean for whether or not it is a ratter. Pass the name to the superclass (Dog) constructor using super, and store the boolean locally.
   d. Override the **speak** method. A Labrador says YAPYAPYAP!
   e. The overridden **averageBreedWeightKG** method should return YORKSHIRE__WEIGHT.
   f. Override the **toString** and the **equals** methods. The toString should contain the name and whether or not the Yorkshire is a ratter. Yorkshires are equal if their names are equal AND their ratter-training status is equal.
4. Finally, create a driver class called DogTest which should only contain a main method:
   a. Try to instantiate a new Dog. What happens? Comment the code out, but be ready to demonstrate and explain what happened to your Lab Instructor.
   b. Instantiate a new Labrador called Hannah who is golden. Print out Hannah's toString, make her speak, and then print out her breed's average weight.
   c. Instantiate a new Yorkshire called Sebastian who is a ratter. Print out Sebastian's toString, make him speak, and then print out his breed's average weight.

3. 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.
    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. Finally, 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.
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: $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix}$. Try adding it to one of the square matrices. Can you catch the exception and print the error for the user?

4. Some more inheritance.

Let's play some more with inheritance. This time, we will work with animals and have a little fun with interfaces too:

1. Let's start by creating the interfaces:
    a. Create an interface called **Terrestrial**. The Terrestrial interface contains a method called run which accepts no parameters and returns nothing.
    b. Create an interface called **Aerial**. The Aerial interface contains a method called fly which accepts no parameters and returns nothing.
    c. Create an interface called **Aquatic**. The Aquatic interface contains a method called swim which accepts no parameters and returns nothing.
2. Create an abstract superclass called **Animal**:
    a. An Animal has an age which is an integer, and which should be stored as a protected instance variable.
    b. Animal should also contain abstract methods called eat and move. Neither method accepts any parameters, and both have void return values.
3. Create an abstract subclass of Animal called **Bird**:
    a. A Bird has a feather colour is passed to the constructor as a String and stored as a String.
4. Create a subclass of Bird called **Ostrich**:
    a. An Ostrich is a Terrestrial being, so the Ostrich class must implement Terrestrial.
    b. Ostrich contains a constructor which accepts an int for the age and passes it to the superclass (Animal) constructor.
    c. When an Ostrich eats it says "Watch me gobble up these lizards yum yum!" (you can print this to the console).
    d. When an Ostrich moves it should run, i.e., call the run method from inside the move method.
    e. When an Ostrich runs, it should say "Look at how fast I lope across the savannah!".
5. Create a subclass of Bird called **Penguin**:
    a. Penguin contains a constructor which accepts an int for the age and passes it to the superclass (Animal) constructor.
    b. A Penguin is an Aquatic being, so the Penguin class must implement Aquatic.
    c. A Penguin is also a Terrestrial being, so the Penguin class must implement Terrestrial.
    d. Penguin contains a constructor which accepts an int for the age and passes it to the superclass (Animal) constructor.
    e. When a Penguin eats it says "I love oily nutritious fish!" (you can print this to the console).
    f. When a Penguin moves it should run and swim, i.e., call the run method and the swim method from inside the move method.
    g. When a Penguin swims, it says "Splash!".
    h. When a Penguin runs, it says "Waddle, waddle, waddle!".

6. Create a subclass of Bird called **Cormorant** (if you've ever walked on the Seawall around Stanley Park, you've probably seen cormorants sunning themselves on a warm rock or flying low and fast just above the surface of the water):
    a. Cormorant contains a constructor which accepts an int for the age and passes it to the superclass (Animal) constructor.
    b. A Cormorant can fly, so the Cormorant class must implement Aerial.
    c. A Cormorant is also a great diver and an Aquatic being, so the Cormorant class must implement Aquatic.
    d. A Cormorant is also a Terrestrial being, so the Cormorant class must implement Terrestrial.
    e. Cormorant contains a constructor which accepts an int for the age and passes it to the superclass (Animal) constructor.
    f. When a Cormorant eats it says "Eat quickly, yum yum!" (you can print this to the console).
    g. When a Cormorant moves it should run and swim and fly, i.e., call the run method and the swim method and the fly method from inside the move method.
    h. When a Cormorant swims, it says "I am a great diver, watch out fish!".
    i. When a Cormorant flies, it says "I must fly fast and low to avoid eagles!".
    j. When a Cormorant runs, it says "Watch me run with my winds wide open for balance!".
7. Make a driver file called BirdFarm that contains a main method. Create an Ostrich, a Penguin, and a Cormorant. Invoke the move method on each Bird.
8. **CHALLENGE**: Instead of storing the Ostrich, the Penguin, and the Cormorant in variables of type Ostrich, Penguin, and Cormorant, store them in variables of type Bird. What is this madness? Can you store them in variables of type Terrestrial? What's happening? We'll find out next week (or take a peek at chapter 10)!

5. You're done! Show your lab instructor your work.