ITSC 1213 - Inheritance Part 2

Introduction

The goal of this lab is to practice working more with inheritance hierarchies.

Concepts covered in this lab:

- Inheritance hierarchies
- Overriding methods

Required files or links

- This lab assumes you have successfully completed the Inheritance 1 lab.

Part A: Overriding methods

1. Open the NetBeans Project you created in the previous lab. Recall the inheritance hierarchy of `Person` classes from the previous lab.

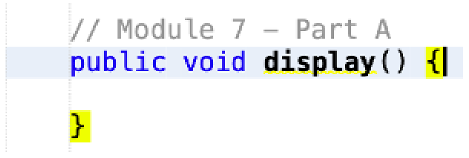
   Both `Student` and `Professor` extends a common base class `Person` and has the methods `getFirstName`, `getLastName`, `getId`, and `display`. We saw how a subclass has access to public methods derived from the parent (super) class. A Student/Professor object was used to invoke the `display()` method even though it was not explicitly added to those classes. We saw the same behavior when this method was called regardless of the specific type of object used to invoke it.

   But as we move down the inheritance hierarchy, we get to more specialized classes. What may have been appropriate general behavior for the super class may no longer be appropriate for the subclass.

   Well, you can just override it, by writing a new method with the same name in the subclass. When Java is compiling the code it can tell that the subclass is overriding the method from the superclass and will instead only use the subclass's version of that method.

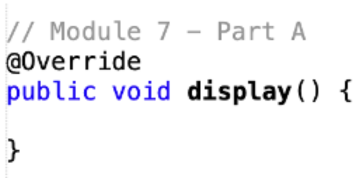
2. Open the `Student` class and add the `display()` method at the end of the class. Why do we need to define it again here? This is because we are not satisfied

with the `display()` method in the `Person` class, which only prints out a person's name and id. We would like this method to display additional information for a `Student` object. To do so, we will override this method by specifying a new implementation.



```
// Module 7 - Part A
public void display() {

}
```

3. If you look to the left of this method, you'll see the yellow light bulb and a black arrow pointing down. If you hover over the lightbulb, NetBeans asks you if it can add an 'override' annotation. You can go ahead and select this and NetBeans will add `@Override` annotation above the method. This notation indicates that this method overrides the `display()` method that is in the superclass, `Person`.



```
// Module 7 - Part A
@Override
public void display() {

}
```

Frequently when we override a method, we don't want to completely replace the super class method. Instead, we would like to do some extra work such as displaying all of the attributes of a student.

4. Now, implement this method so that it will print out a student's name, their id, major, gpa, number of credits and the list of classes that the student is enrolled in. The keyword `super` can be used to tell the system that you want to use something from the parent class. It is used to call methods from the parent class, and to access the parent class constructor. To display the name and id, you can simply call the display method of the superclass, by using the reserved keyword `super`:

```
super.display();
```

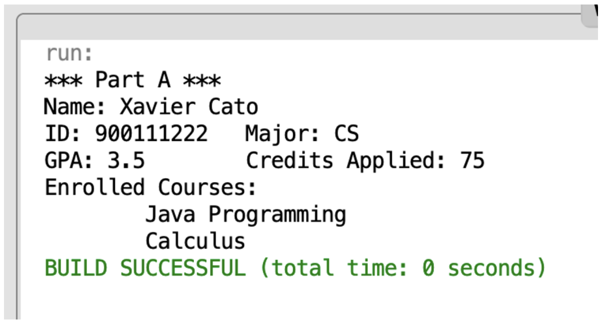
Note that if you want to format the display independently of how the display method of the superclass works you must use the getter methods since we set those fields to have private access. Make sure to add additional statements to print out all of the student details.

```
// Module 7 - Part A
@Override
public void display() {
    System.out.println("Name: " + this.getFirstName() + " " + this.getLastName());
    System.out.println("ID: " + this.getId() + "\tMajor: " + major);
    System.out.println("GPA: " + gpa + "\tCredits Applied: " + credits);
    System.out.println("Enrolled Courses:");

    for (int i = 0; i < enrolledCourses.size(); i++) {
        System.out.println("\t" + enrolledCourses.get(i));
    }

}
```

5. Let’s test to make sure our added method in the subclass works as intended.

```
Student s1 = new Student("Xavier", "Cato", 900111222, "CS", 3.5, 75);
s1.setTransfer(true);
s1.setBalance(100);
s1.addCourse("Java Programming");
s1.addCourse("Calculus");

s1.display();
```

Since we already have a call to this method in the main method of the main class file you should be able to run this project, and see an output similar to this:



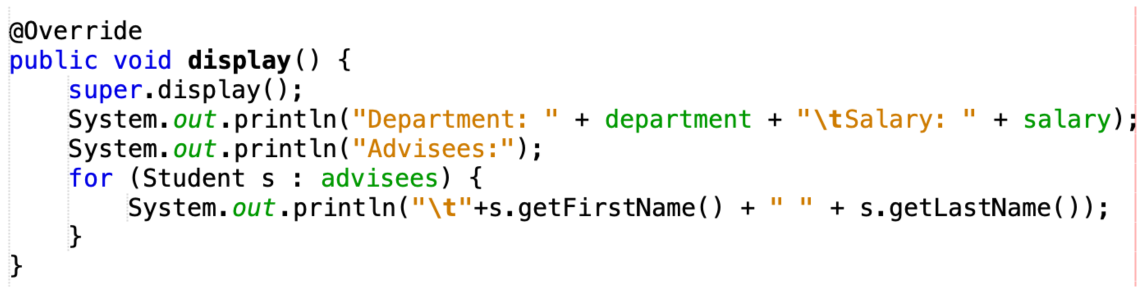
```
run:
*** Part A ***
Name: Xavier Cato
ID: 900111222   Major: CS
GPA: 3.5        Credits Applied: 75
Enrolled Courses:
        Java Programming
        Calculus
BUILD SUCCESSFUL (total time: 0 seconds)
```

6. Run your project and fix any bugs you encounter. When you are satisfied with your program, show your work to the instructional team to get checked off and proceed to Part B.

Part B: Overriding methods for Professor

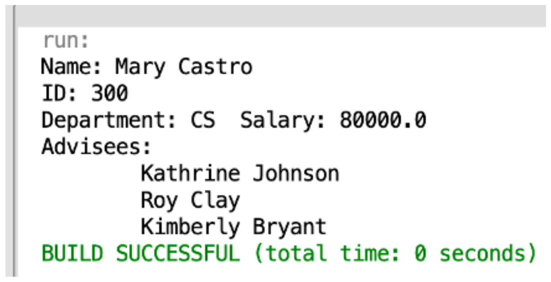
1. Follow the same procedure we followed in Part A to add the display method to the `Professor` class.

2. Implement this method so that it will print out a Professor’s name, their id, department, salary, and the list of advisees of this professor.



```
@Override
public void display() {
    super.display();
    System.out.println("Department: " + department + "\tSalary: " + salary);
    System.out.println("Advisees:");
    for (Student s : advisees) {
        System.out.println("\t"+s.getFirstName() + " " + s.getLastName());
    }
}
```

3. Again, we shouldn't need to change anything in the main class to test this method. Test your added method by running the project you should see an output similar to this:



```
run:
Name: Mary Castro
ID: 300
Department: CS  Salary: 80000.0
Advisees:
        Kathrine Johnson
        Roy Clay
        Kimberly Bryant
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Fix any bugs you encounter. When you are satisfied with your program show your work to the instructional team to get checked off and proceed to Part C

Part C: The Parent of all classes

In Parts A and B of this lab we saw how overriding a superclass method provides a way to change the functionality that would be inherited by a subclass. We also saw that overriding a superclass method may involve a complete replacement of functionality or the subclass may invoke the superclass method to provide part of the functionality.

In Java every new class inherits from the universal superclass "Object". This is how every class has a `toString()` method: since Object has a `toString()` method, then 'children' of Object inherit a toString() method, the children of children of Object inherit a `toString()` method, and so on. So every class 'automatically' gets a `toString()` method by inheritance.
But what does this `toString()` method actually do? The `toString()` method is intended to create a string which describes the object. By default, `toString()` creates and returns a reference to a String which looks like:

```
ClassName@4edf23fa6cc3
```

How does this 'describe' the object? The answer is that it doesn't do a very good job. It prints the name of the class that the object came from, and some other alphabetic and numeric characters related to where the object is stored in memory, but nothing else.

We can fix that situation. We can write our own `toString()` method, and have it create and return a reference to a string of our choosing. This way, we can customize the `toString()` method so it is well suited to the objects of our 'child' class. As we've seen, doing this is called ***overriding*** the inherited method, and if the method that we write has the same name, parameters, and return value, then our method will be called rather than the one that we inherited. Therefore, when we want to write a method that returns a reference to a string which better represents an object in our class we just write a `toString()` method. You can pretty much define the `toString()` method as you like, as long as it has no parameters, returns something of type String, and has the name toString.

1. In the student class override the `toString()` method to return a short description of a `Student` instance. For example, this might look like this:

```java
@Override
public String toString() {
    return "Student - "+this.getFirstName() + " " + this.getLastName();
}
```

2. To test this, we can simply pass in a student reference to the `println` method which will call the `toString` method of that object.

```java
Student s1 = new Student("Xavier", "Cato", 900111222, "CS", 3.5, 75);

System.out.println(s1);//this compiles to s1.toString()
```

3. Repeat the same process of overriding the `toString` method in the `Professor` class

```java
@Override
public String toString(){
    return "Professor - "+ this.getFirstName() + " "+ this.getLastName();
}
```

4. Now test it in the main method to make sure it executes as expected

```java
Professor prof1 = new Professor("Mary", "Castro", 300, "CS", 80000);
System.out.println(prof1);
```

5. In addition to the `toString()` method every class inherits the `equals()` method of the `Object` class. An object is characterized both by its identity

(location in memory) and by its state (actual data). The default `Object.equals()` implementation compares only the references as it uses the == operator which compares only the identities of two objects (to check whether the references refer to the same object). But the `equals()` method defined in `java.lang.Object` can be overridden to compare the state as well. When a class defines an `equals()` method, it implies that the method compares state. When the class lacks a customized `equals()` method (either locally declared or inherited from a parent class), it uses the default `Object.equals()` implementation inherited from `Object`.

When you override any method you must match the method name, return type, and parameter types ***exactly***. For the equals method, it must be this:

```
public boolean equals(Object other)
{
   // Logic here
}
```

Notice that the parameter type for the equals method is `Object` - ***it must be Object*** or you will have ***overloaded*** equals instead of ***overriding*** it. The effect is that, depending on the type of the parameter being passed to equals, sometimes your equals method will execute and sometimes the one in `Object` (performing strict reference equality) will be executed which, you recall, performs reference equality (identity) not logical equality.

Now if we want to override this behavior for both of the `Student` and the `Professor` so it is based on a more logical check for equality it would make sense to have this based on the id attribute as it is used to save a unique identifier value. We can certainly override the equals method in each of the Student and Professor classes but we would quickly realize that these methods would be identical. And since both classes are subclasses of the Professor class we can utilize inheritance here and just define this behavior once in the parent class.

6. Open the `Person` class and override the equals method

```java
@Override
public boolean equals(Object other){
     if (other == null) {
        return false;
    }

    if (this.getClass() != other.getClass()) {
        return false;
    }

    return this.getId() == ((Person) other).getId();
}
```

Notice how we are verifying that the type of the object passed in as the argument passed is in fact a `Person` object before we check its id.

Failing to do this can result in a `ClassCastException` being thrown if this method happens to be called with different types of object.

7. Now, go back to the main method to see how this method behaves. Assuming you still have the `Student` objects we created in the previous lab we can compare `s1` and `s4`. This should return false as these represent two different students.

```java
System.out.println(s1.equals(s4));
```

8. If we create a `Student` object with the same id we have for another student we can see how this method identifies that those two objects are equal.

```java
Student s5 = new Student("Xavier", "Cato", 900111222, "CS", 3.5, 75);

System.out.println(s1.equals(s5));
```

9. And if we use a different type of object we can see that the method returns false as they are not of the same class.

```java
System.out.println(s1.equals(prof1));
```

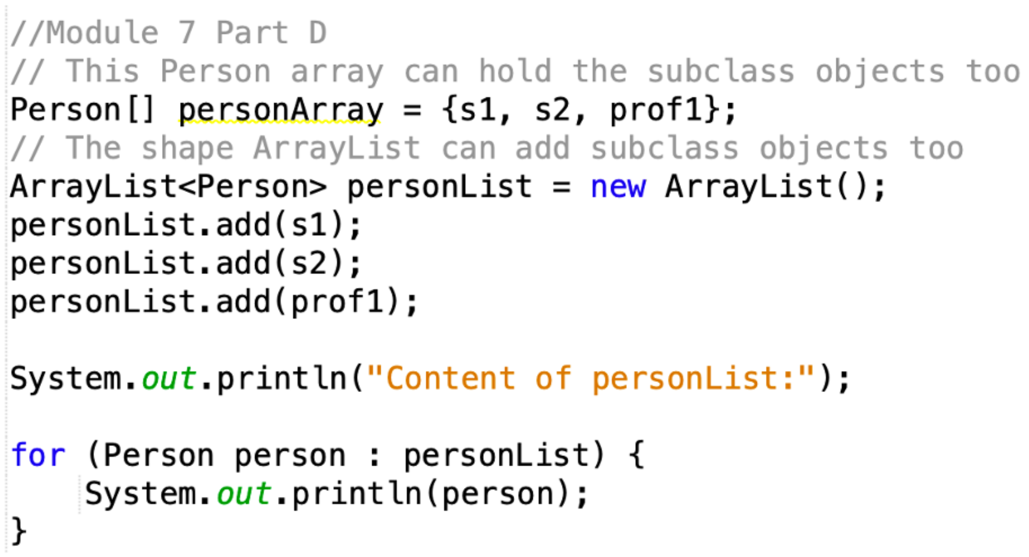
10. Verify the equals method works as expected and fix any bugs you encounter. When you are satisfied with your program, show your work to the instructional team to get checked off and proceed to Part D.

Part D: Superclass Arrays and ArrayLists

In section 9.5.3 we saw how using inheritance hierarchies, we can create arrays and ArrayLists using the superclass type and put in values that are of the subclass type. This can be very useful which you will appreciate more in the next lab as we dive more into polymorphism.

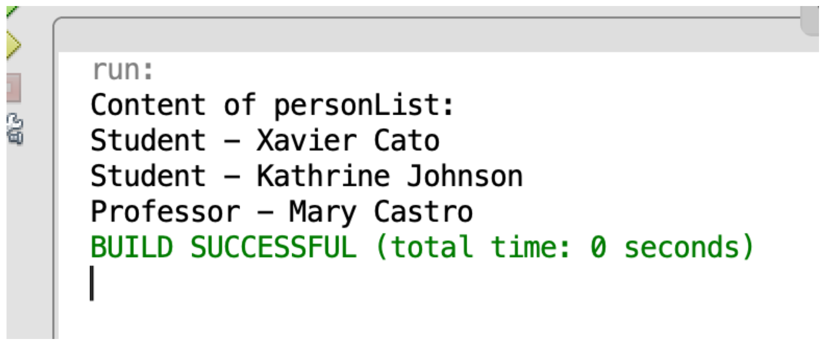
In this part we will just practice how we can create a `Person` array or ArrayList that can hold any objects of the `Person` subclasses.

1. In the main method create an Arraylist of type `Person`
2. Add some of the objects of `Professor` and `Student` we have
3. Iterate calling the `toString()` method, what method is being called?



```java
//Module 7 Part D
// This Person array can hold the subclass objects too
Person[] personArray = {s1, s2, prof1};
// The shape ArrayList can add subclass objects too
ArrayList<Person> personList = new ArrayList();
personList.add(s1);
personList.add(s2);
personList.add(prof1);

System.out.println("Content of personList:");

for (Person person : personList) {
    System.out.println(person);
}
```

Your output might look like this:



```
run:
Content of personList:
Student - Xavier Cato
Student - Kathrine Johnson
Professor - Mary Castro
BUILD SUCCESSFUL (total time: 0 seconds)
```

4. Verify that this works as expected and fix any bugs you encounter. When you are satisfied with your program, show your work to the instructional team to get checked off for this lab.