

ITSC 1213 - Abstraction and Interfaces

Introduction

The goal of this lab is to practice working with abstract classes and methods and implementing interfaces.

Concepts covered in this lab:

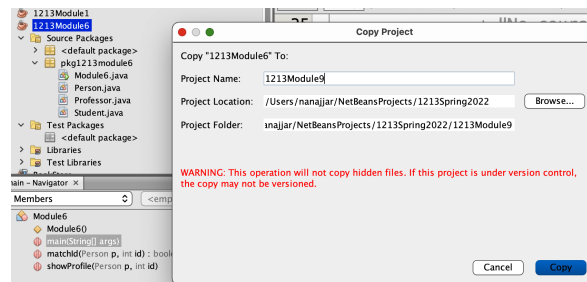
- Defining abstract classes and abstract methods in superclasses
- Creating concrete subclass versions of a superclass abstract methods
- Implementing the Comparable interface
- Modifying a class to specify how Comparable will sort members of that class

Required files or links

- This lab assumes you have successfully completed the previous lab.

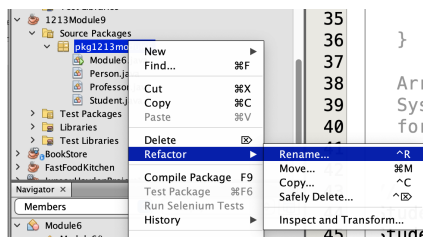
Part A: Make a copy of an existing NetBeans project

- 1) This lab builds on the inheritance labs we've been working with. To get started we want to make a new copy to use and refactor for this lab. Right-click on **your inheritance** select Copy and specify a new name for the project (e.g., Module9)



You should see this new copied project appear in your project window.

If you had your code inside a package you might want to update the name to reflect the new project. The best way to do that is using the refactor functionality in NetBeans which will update any references we have to the old names inside of this project. To do that right click on the package/class name select refactor then Rename. Update then name in the input box and select Refactor.



- 2) In the project we already have created some classes that represent things (objects) that can be relevant to a university system or application. More specifically, these classes represent different roles people play in this system. Professor and Student are examples of those roles. When do we have just a person in this system that does not have some specific role?" If someone asks you what is your role at UNCC? you don't reply with "a person" unless you are trying to be purposefully obtuse! The point we are trying to make is that we shouldn't allow anyone to create a Person. Person is an *abstract*, rather than a specific, type of object. So, go on to Part B, and make that happen. (There are no points for Part A).

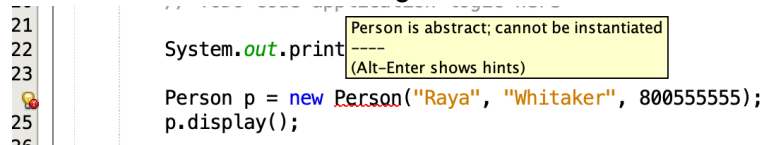
Part B: Make Person into an abstract class

Abstract classes allow you to define the parent class of a new hierarchy without having to worry about the user actually instantiating the parent. For instance, you could create an "Animal" class just to act as the basis for "Dog", "Cat", "Sheep" and so on, even defining some functionality in "Animal", but at the same time preventing the user of your hierarchy from trying to create an "Animal" object (which after all wouldn't make much sense -- you never encounter an abstract "animal" in the real world; only particular kinds of animals).

To make it so that no-one can create a Person object, we simply need to make that class abstract.

- 1) First, in the class definition, add the keyword `abstract` before or after `public`.
- 2) Now, we can see that this change has created an error in our main method in **the main class** file (this assumes we have the statements to instantiate a Person object from the last lab).

By definition we cannot instantiate an abstract class. Java forces you to create *concrete* methods for abstract classes. If you look at the lines where we instantiated any Person objects you'll see that NetBeans is now telling us that Person cannot be instantiated:



```
21
22
23
24
25
26
System.out.print
Person p = new Person("Raya", "Whitaker", 800555555);
p.display();
```

You can comment or delete those statements since we don't need them anymore.

- 3) Now, the question is which methods in Person should be abstract and which ones shouldn't. How do you know?

When thinking about the methods inside of an abstract class, you should only decide to make a method abstract if you feel that every subclass must have its own version. If the method does something that would be the same for all the subclasses it's OK for it to be concrete and defined in the superclass, in fact it shouldn't be abstract in that case.

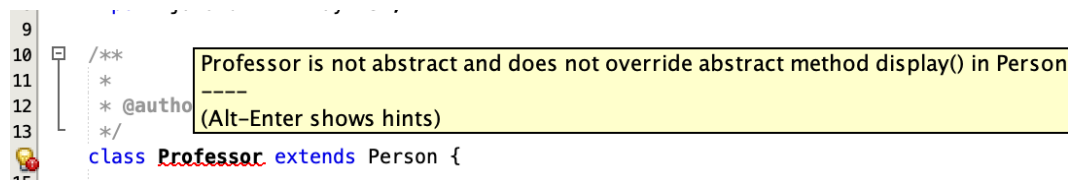
Here is a table of all the methods inside Person. Write yes beside the ones you think should be abstract (the ones where the behavior of the method should be different for every sub-class of this class):

Person methods	Should it be abstract? (Yes or No)
public Person(String firstName, String lastName, int id) [constructor]	
public String getFirstName()	
public String getLastName()	
public int getId()	
public void display()	

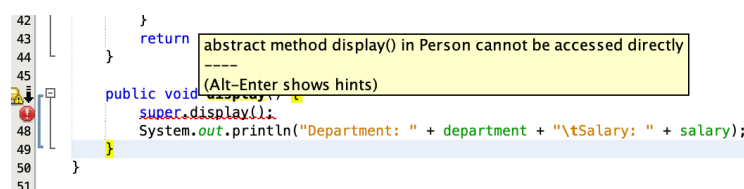
- 4) Hopefully, you only wrote yes beside the `display(...)` method. This is the one that needs to be different for each person (because each person's details are different). So, to make that method abstract, add the keyword `abstract` before or after `public` in the method header.

public abstract void display();

- 5) Now, we already have a version of `display(...)` in each of our sub-classes (Student, Professor) which is a required method since both of these classes are not abstract classes and must override all abstract methods that are in the super class. If we try and delete the `display` method in the Professor class we will see that NetBeans is telling us that we have extended an abstract class, but we haven't yet implemented the abstract methods within, namely the `Person display()` method.



So, we know we need to keep the `display` method; but we still have an error in the Professor class. Why do you think that is? How can we fix it?



Hovering over the line with the error we see that NetBeans is telling us that the abstract method `display()` cannot be accessed directly. In order to fix this we need to replace this


statement with code that accomplishes the same logic (behavior/functionality) the display() method did in the superclass.

Making the display method abstract gives us the ability to implement a unique and custom behavior of this method for each subclass we create.

Update this method to display the name and ID of a Professor:

```
public void display() {
    System.out.println("Name: " + this.getFirstName() +
        " " + this.getLastName() + "\tID: " + this.getId());
    System.out.println("Department: " + department + "\tSalary: $" + salary);
}
```

Now, run your program and fix any bugs you find.



```
Output - 1213Module8 (run) x
*** Part C ***
Name: Mary Castro      ID: 300
Department: CS Salary: $80000.0
```

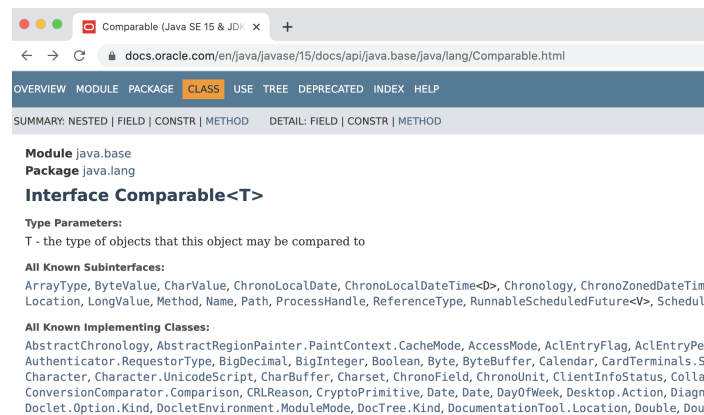
- 6) When your program works and you are satisfied with the result, show your work to the instructional team to get checked off and proceed to Part C.

Part C: Implement the Comparable interface to sort students

- 1) What if we wanted to sort students by GPA, so that we can display students in order, from lowest to highest GPA? We could program this ourselves, but Java already has a Comparable interface.

If we implement that interface, and write a single method that tells Java how to compare students for GPA, we don't have to iterate through our list and keep track of stuff to do the sorting, Java will use our compare students method and do the sorting for us. How awesome is that?!

That's what we'll do here. To get started, google the documentation for the Java Comparable interface. You should see something that looks like this:



- 2) One thing to notice right away is that this interface makes use of generics – note the angled brackets and the T symbol. This syntax says that the Comparable interface can be applied to classes of any type: String, Turtle, Picture, Person, Student, Round, etc. The use of generics (i.e. not being specific about the type of object until you actually implement the thing) is what makes interfaces so powerful. The Comparable interface can be implemented for any class that already exists, or any class you may create in the future.
- 3) If you scroll down, you'll notice that there is one method - `compareTo(...)` - that you must implement if you want to use this interface:

The screenshot shows the 'Method Summary' and 'Method Details' for the `compareTo` method in the `Comparable` interface. The 'Method Summary' table has three tabs: 'All Methods', 'Instance Methods', and 'Abstract Methods'. The table lists the method `compareTo(T o)` with a return type of `int` and a description: 'Compares this object with the specified object for order.' The 'Method Details' section shows the signature `int compareTo(T o)` and a description: 'Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.'

Modifier and Type	Method	Description
int	<code>compareTo(T o)</code>	Compares this object with the specified object for order.

Method Details

compareTo

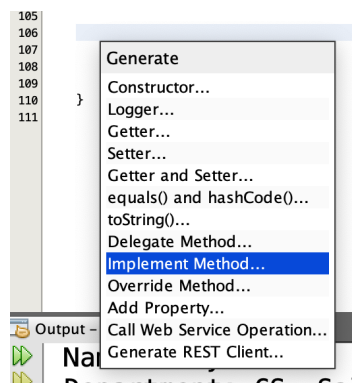
`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

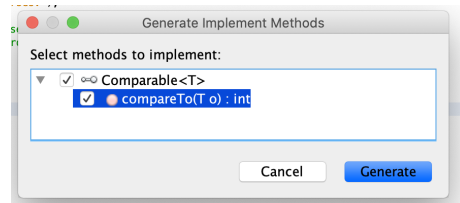
- 4) If we are going to compare students using each student GPA value, we'll need a getter for the GPA field which we already have but we still need to add `implements Comparable<Student>` to the class definition after `extends Person`.
- 5) You know from doing the prepwork that if you use a Java interface, you need to add the methods in the interface to your class and actually implement them. But since Student is a subclass of Person, you might be wondering, do we want to implement the `compareTo(...)` method in Person, or do we want to implement it in all of the subclasses? In this case, since we want to sort students based on the `gpa` field, we'll implement the interface in the student class.

In some situations, we might want to sort person objects based on the `id` field, and since `id` has been defined in Person, in this case we would want to implement the Comparable interface there, rather than having to implement it multiple times in each of the subclasses.

- 6) To update the Student class to implement the interface, right-click inside your Student class, select Insert Code, and then, in the list that pops up, choose Implement Method...



- 7) You'll then see a dialog that lists all the methods from all the interfaces that are in use. In this case, it is just the Comparable interface, and just the `compareTo (. .)` method:



- 8) Select `compareTo` and click `Generate`.
- 9) Now you'll see the following in your code:

```
@Override
public int compareTo(Student o) {
    throw new UnsupportedOperationException("Not supported yet."); //To change
}
```

Note, depending on the exact version of Java that you are using, the generated code might give you `Object t` as the parameter, instead of `Object o`. Inside the curly braces (body of the method) we don't need the `throw new...` statement about throwing the exception so delete that whole line.

Side note: that when we made `Student` implement `Comparable` we specified the type of object that will be used (`Student`). If we did not specify the type when we go to override the `compareTo` method it would use `Object` for the parameter type and we would need to cast that to `Student` inside of the method to operate on it.

```
@Override
public int compareTo(Object o) {
    Student otherObject = (Student) o;
    System.out.println(otherObject.getId());

    return 0;
}
```

```
@Override
public int compareTo(Student o) {
    System.out.println(o.getId());

    return 0;
}
```

- 10) Right now, there's a missing return error but it will go away when we complete this method. If you carefully read the documentation for the `Comparable` interface you'll note that it compares a passed-in object (let's call it 'ot') to `this` object (let's call that 'th'). If by whatever means you compare these objects, `th` is less than `ot`, the `compareTo (...)` method should return a negative number (usually it's -1). If `th` is greater than `ot`, the method should return a positive integer (usually 1). If they are equal, the method should return 0.

Here is pseudocode:

- get `THIS` student's GPA (we'll refer to it as `sGPA`).
- get the other student's GPA (we'll refer to it as `oGPA`).
- compare `sGPA` to `oGPA` and return 1, -1 or 0, depending on the result
 - o return 1 if `sGPA` is greater than `oGPA`
 - o return -1 if `sGPA` is less than `oGPA`
 - o return 0 if they are equal

- 11) So, you can see what your method is doing, it would be a good idea to add some print statements to it. Try adding a statement like this at the beginning of the `compareTo` method:

```
System.out.println("Students GPA: " + sGPA + " and " + oGPA);
```

- 12) Now, in order to test this `compareTo` method, open the main class, and add the following line to the end of the `main` method:

```
int compareStudents = s4.compareTo(s1);  
System.out.println(compareStudents);
```

If you now run this, you should see something like this if the first two students are the same:

```
Students GPA: 3.5 and 3.5  
0  
BUILD SUCCESSFUL (total time: 0 seconds)
```

or like this, if they are different:

```
Students GPA: 3.0 and 3.5  
-1  
BUILD SUCCESSFUL (total time: 0 seconds)
```

After you're sure your `compareTo(...)` method is working properly, you can delete the test lines if you like.

- 13) Now, it's fine to be able to explicitly compare the GPAs of two students. But what is really cool about this is the combination of having students being in an `ArrayList` (which implements the `Collection` interface) and your `Student` implementing the `Comparable` interface. This means that you can take an `ArrayList` of `Student` objects and tell it to sort itself! And it can because `Students` now know how to compare themselves to each other!

Let's make use of this functionality to make a list of students display from lowest to highest GPA.

- 14) Now, in the `main(...)` method, let's create an `ArrayList` of students and add 5 five students. We already created some in the last lab so we can just add those objects.

```
ArrayList<Student> students = new ArrayList();  
students.add(s1);  
students.add(s2);  
students.add(s3);  
students.add(s4);  
students.add(s5);
```

- 15) To compare and sort this list of students add the following code:

```
Collections.sort(students);  
  
for (Student s : students) {  
    System.out.println(s.getGPA());  
}
```

If you notice an error where you are referencing the Collections class it can be resolved by importing.

```
import java.util.Collections;
```

The first line in this code snippet tells the ArrayList of students to sort itself. Because Student implements the `compareTo()` method and makes students be sorted by their GPA, this will sort the vehicles in that order and rearrange how the students are stored in the array list. The for loop that follows simply displays the GPAs from low to high.

- 16) Run this project a few times to make sure that it works. Fix any bugs you find.
- 17) When your program works and you are satisfied with the result, show your work to the instructional team to get checked off and proceed to Part D.

Part D: Implement a Custom Interface

We've seen that when a class implements an interface, it is giving a guarantee that it can perform a certain functionality, as defined by the interface it implements. You'll find that the idea of an interface is actually very central to software development and engineering in general. When you're asked to implement a set of methods to perform some specific task, that's implementing an interface. Often when working on a group project, a good approach is to split the work into parts that will be integrated together at the end. In order to allow work to be done in parallel, it is important to establish what each part will accomplish and how it will interact with other parts so that they can be merged together without issue. Establishing what each of these parts will do and how they interact with other parts is essentially treating each part as an interface. Using interfaces is all about not knowing the actual implementation, but instead utilizing the input-to-output, defined behavior given by the interface; implementing an interface to specification like you are asked for assignments and projects is about making sure the program you write under some interface gives the correct output for all inputs.

In this part you are asked to create a class that implements the following interface:


```

public interface UniversitySpecification {

    /**
     * setUp adds the initial personnel (students and professors) of a university
     *
     * @param personnel - the initial list of personnel of this university
     */
    public void setUp(ArrayList<Person> personnel);

    /**
     * getStudents get students from personnel list
     *
     * @return a list of students
     */
    public ArrayList<Student> getStudents();

    /**
     * getProfessors get professors from personnel list
     *
     * @return a list of professors
     */
    public ArrayList<Professor> getProfessors();

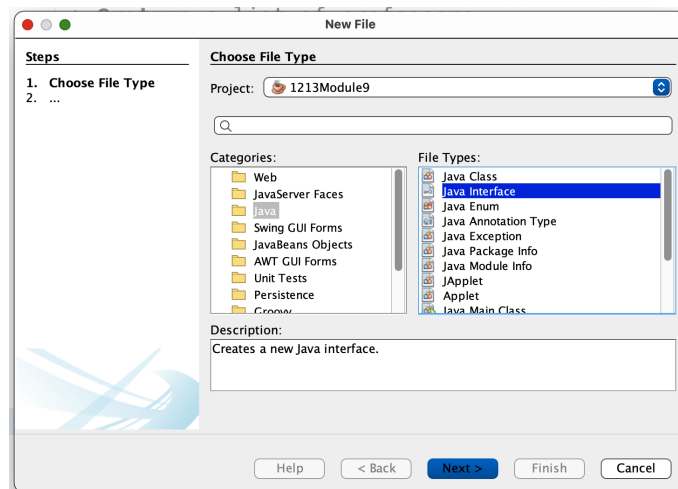
    /**
     * add a new student to university personnel
     *
     * @param s - new student to add
     */
    public void newStudent(Student s);

    /**
     * add a new professor to university personnel
     *
     * @param p - new professor to add
     */
    public void newProfessor(Professor p);

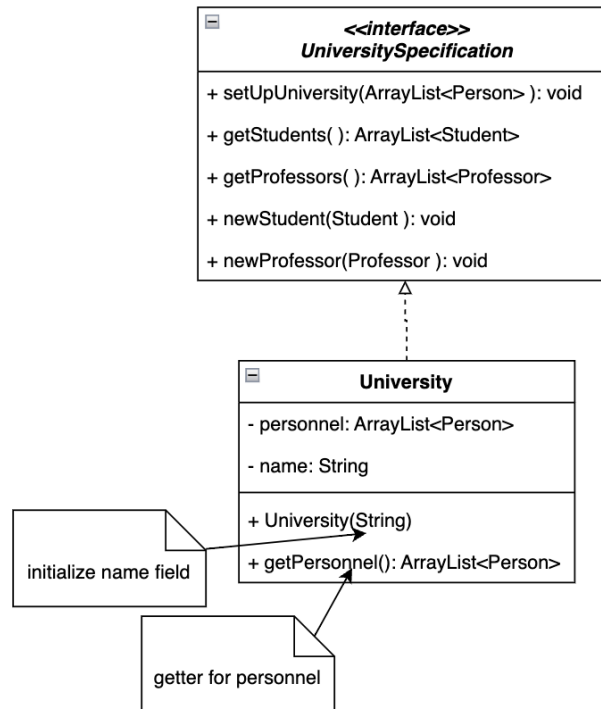
}

```

- 1) While we haven't created any interfaces before, the process is similar to creating a new class (File → New File ...).



- 2) Name the new interface UniversitySpecification and copy and paste the code from [this code snippet](#) into this file. Make sure you include the correct package declaration at the beginning of the file to match what you have in your project. Examine the code you just copied to get a sense for what it is doing. This file contains general specifications for a program that could be a part of a management system for an educational institution (e.g., university).
- 3) Now you need to create a new class called University that is based on the following UML diagram. Notice that this class implements this UniversitySpecification interface. Don't worry about testing your code until you are done with the implementation.



- 4) When you are ready to test your code, create a new class called `UniversityDriver` and copy and paste the code from [this code snippet](#) into this file. Make sure you include the correct package declaration at the beginning of the file to match what you have in your project. Explore the code you just copied to get a sense for what it is doing. Make sure everything compiles and then run this file. Review the output and verify it is what you expect it to be.

```

Output - 1213Module9 (run) x
run:
**** Testing University class implementation ****
University setup works as expected
University getStudents method works as expected
University getProfessors method works as expected
University getStudents method works as expected
University getProfessors method works as expected
BUILD SUCCESSFUL (total time: 0 seconds)
  
```

- 5) When your program works and you are satisfied with the result, show your work to the instructional team to get checked off for this lab.

Bonus

- Update the professor class so that it implements the `Comparable` interface to sort Professors based on some criteria.