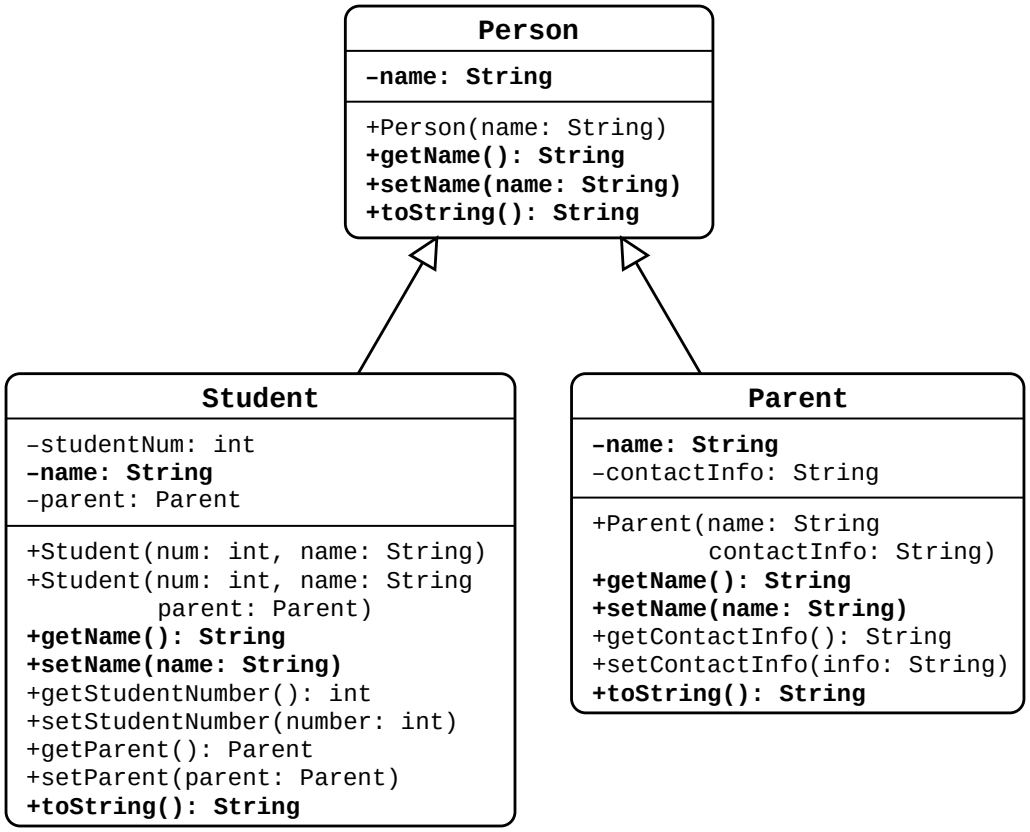
PBL: Inheritance



Follow the steps below to implement these

1. Implement the `Person` class as described by the UML diagram shown above, and create a separate class to test the basic functionality. For the `toString` method, simply return the field `name`. As the `toString` method is overriding the implementation of `toString` we inherit from the `Object` class, it is best practice to prefix the `toString` method with the `@Override` annotation, like so:

```
@Override
public String toString() { … }
```

   Once you have completed both the code for the `Person` class, and also the code to test the class, show your code to the teacher.

2. Implement the `Parent` class as described by the last UML diagram. Either write a new class to test its functionality, or update the class you used to test the `Person` class. If you update the class that tests the `Person` class, do not remove the code that tests that class, just cleanly add the new test code. Override the `toString` method of the `Person` class in the `Parent` class so it returns a string that includes both the `name` field and the `contactInfo` field in a format that you think makes sense for printing to the screen. Again, it is best practice to annotate this method with the `@Override`. Once you have completed the code for both the `Parent` class, and the code that tests it's functionality, show your code to the teacher.

3. Implement the Student class as described by the last UML diagram. Again, you can write a new class to test its functionality, or add it to your previous test class – but do not remove your previous test code. Override the `toString` method of the `Person` class in the `Student` class so it returns a string that includes both the `name` field and the `studentNum` field in a format that you think makes sense for printing to the screen. Include the `@Override` annotation. Once you have completed the code for both the `Student` class, and the code that tests it's functionality, show your code to the teacher.

Summary of Inheritance for Code Re-Use

The example we have coded thus far shows how inheritance helps programmers leverage a hierarchy of types in order to avoid duplication of code. In the real world, as well as in our example code, both a `Student` is a type of `Person` and a `Parent` is a type of `Person`. When we define the information (the *fields*) that we need to associate with a `Person` and the functionality (the *methods*) we implement to operate on a `Person`, the other subtypes can ***inherit*** that information and functionality from the super class rather than duplicate it.

Inheritance and Polymorphism

Polymorphism is when subclasses that belong to the same superclass behave differently from one another. This adds flexibility to inheritance. In order to fully understand what polymorphism is, we will continue with our school registry example.

Sorting Students versus Sorting Parents

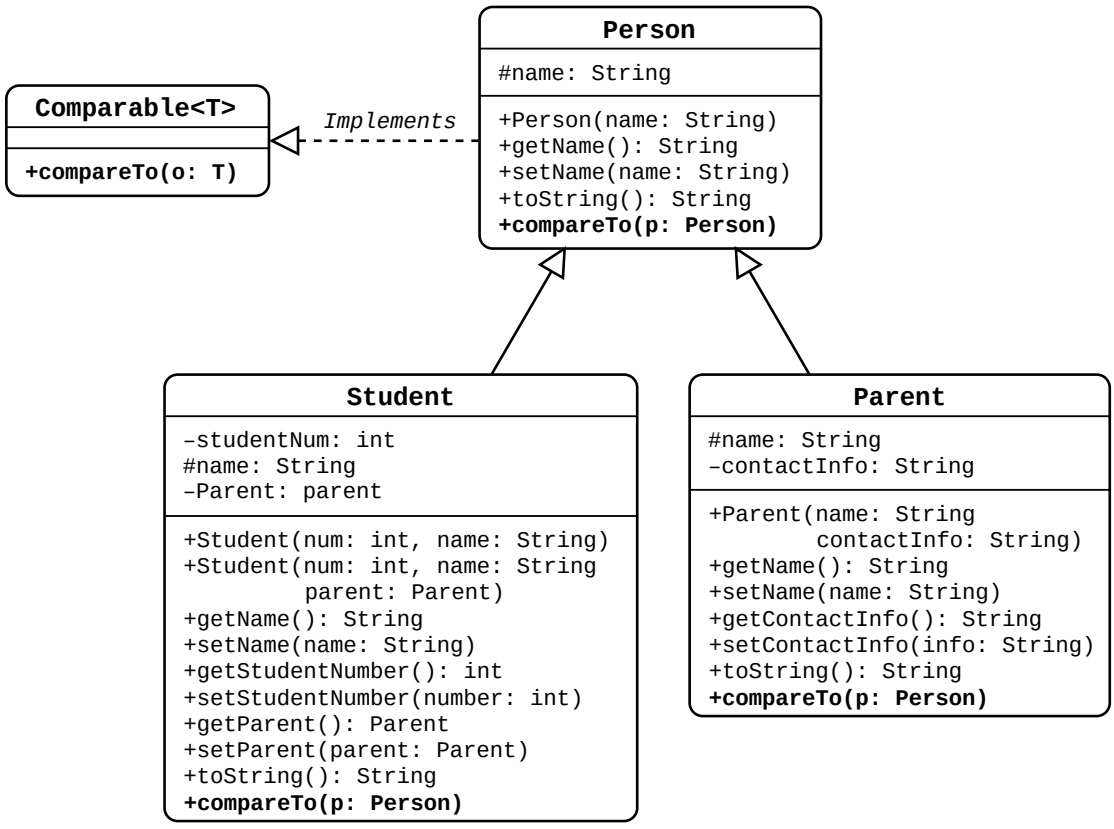
A requirement of the school registry software will be to sort lists of students according to their student numbers, and sort a list of parents by their name. The standard Java class `Arrays` contains methods that can sort arrays. Given an array of type `Student`, defined by `Student[] students`, we can sort the array by calling:

```
Arrays.sort(students);
```

As objects of type `Student` contains multiple fields, the question arises: how does this `sort` method know how to sort `students`? The answer to this question is that the `sort` method will call a method named `compareTo` that you must define in any class that you wish to be able to sort. The `sort` method accepts any object that `implements` the `Comparable` interface. Just for your reference, the code for `interface Comparable` is given here:

```
1 package java.lang;
2 import java.util.*;
3
4 public interface Comparable<T> {
5    public int compareTo(T o);
6 }
```

So we will update our `Person` class to implement the `Comparable` interface. There is only one method in this class. Here is the updated UML class diagram for our project, with the new feature in bold font.



4. Change the declaration of class `Person` to:

```
public class Person implements Comparable<Person>
```

   Then write the method `compareTo` that will return:

   - a **negative** `int` if the current `Person`'s name is lexicographically precedes the parameter `Person`'s name
   - **zero** if the current `Person`'s name is exactly the same as the parameter `Person`'s name
   - **positive** `int` if the current `Person`'s name is lexicographically follows the parameter `Person`'s name.

   Hint: the implementation of method `compareTo` should be extremely short to write, as the `String` class implements interface `Comparable<String>`, and thus implements a `compareTo` method. You will probably want to check the documentation for the `String` class `compareTo` method online and call this method to compare the `name` field of `Parent`. There's little sense in writing the code to compare those strings yourself.

   Once you have completed the code for both the `compareTo` method in the `Person` class, and the code that tests it's functionality, show your code to the teacher.

5. In the `Student` class, override the `compareTo` method inherited from the `Person` class so that `Student` objects will be sorted in ascending order by student number (the field `studentNum`). The `Student` class extends the `Person` class, and the `Person` class is already implementing the `Comparable<Person>` interface, so we cannot have the `Student` class implement `Comparable<Student>`. Also, we cannot change the type of parameter when we are overriding. Therefore, the `compareTo` method in the `Student` class will still need to take in a parameter of type `Person`. Prior to checking the student number, we will first need to check if the parameter is of type Student or not. This is accomplished by the expression: `p instanceof Student`. This expression will return `true` if the variable `p` is of type `Student`.

   So, if the parameter `p` is of type `Student`, return:
   `+1` if `studentNum` of the current `Student` is greater than `studentNum` of the parameter `Student`
   `–1` if `studentNum` of the current `Student` is less than `studentNum` of the parameter `Student`.

   If the values for both `studentNum` are equal, or if the parameter `p` is not of type `Student`, then call the `compareTo` method in the superclass (the `Person` class) to sort the `Student` based on their name. You may need to look up online how to call a method from the superclass, and if your research fails, perhaps ask your teacher or classmate for a hint. Just remember: in most cases you will learn a lot more if you make your own best effort before you elicit assistance.

   Once you have completed the code for both the `compareTo` method in the `Student` class, and the code that tests it's functionality, show your code to the teacher.

6. Obtain the teacher's short test class to test your code, and show the results to your teacher.

Summary of Inheritance

In the first part of this document, we examined how inheritance was able to aid programmers in avoiding code duplication. In the latter part of this document, we showed how ***polymorphism*** – the ability of objects of a common super type to behave differently based on its subtype – prevents inheritance from overly restricting the subtypes.

We have completed our discussion of what is called the ***four pillars of object-oriented programming***:

- ***abstraction*** – hiding complexity
- ***encapsulation*** – bundling related information (*fields*) and operations (*methods*) together in one place
- ***inheritance*** – allowing extension of a class' functionality
- ***polymorphism*** – allowing objects to behave differently based on the object type

We have learned the fundamentals of object oriented programming. Further practice will help to solidify our understanding and software engineering abilities.