Java Class Model

The purpose of this lab is to introduce the basics of writing object-oriented code in Java. The lab may appear simplistic in nature, but it is important that the *concepts* of object-oriented coding are grasped correctly, as they will provide a foundation for the rest of the course.

Exercise 1 – Creating a Class

In this exercise, write the `Person` class with a class attribute of type `Address` along with class attributes that store relevant information about the person. It is important that you note that this class follows the concept of *encapsulation*, which is a fundamental part of object-oriented design.

Skeleton classes named `Person` and `Address` have been provided. The `Person` class will model and store basic characteristics of a person. Address data will be handled by a class named `Address`.

Follow the accessibility rules-of-thumb (attributes are private and methods are public). Details include:
A class called `Address`:

- `Address` has 3 attributes: street, city and zip, all of type `String`
- `Address` has a 3-argument constructor to initialize the attributes
- There are 3 `get` methods to retrieve the attributes, e.g., `getStreet()`

The class `Person`:

- `Person` has 3 attributes: name of type `String`, age of type `int` and address of type `Address`
- A 3-argument constructor, to initialize the name, age and address attributes
- A `setName(String name)` and a `getName()` method to set and get the name attribute
- A `setAge(int age)` and a `getAge()` method to set and get the age attribute
- A `toString()` method that returns a textual representation of the class. The `toString()` method is inherited from the `Object` class. We will override it here so that a person is appropriately presented as a string. Note that this method is invoked automatically when the class is used in a string context, e.g., concatenation or as an argument to `System.out.println()`.

Exercise 2 – Test the Person Class

All programs need a starting point. For standalone applications, this is the `main` method. Each class can have a `main` method that can be used as a unit tester, even if it is not the starting point for the entire application.

You could also write a class that functions as a test driver. The purpose of this exercise is to write a driver that starts up the program and tests the `Person` class created in the previous exercise.

1. Write a `main` method for the `Person` class that creates an instance of `Person` and an `Address` by invoking the non-default constructors.

2. Test the class by causing the `toString()` method to be invoked.
   Hint: pass the instance as an argument to `System.out.println()`.

Exercise 3 - Extend the Person Class

All fully-functional object-oriented languages support inheritance. In this exercise, inherit from the `Person` class to create two more classes that are slightly more unique and have different functionality from one another.

1. You are provided two starting files with which to make two subclasses of `Person`: `Employee` and `Contractor`.

2. `Employee` class details:

   a) A `salary` attribute of type `double`.

   b) Write a constructor that takes all the arguments required to construct the `Person` class as well as one to set `salary`.

   c) Add `setSalary(double salary)` and `getSalary()` methods.

3. `Contractor` class details:

   a) Two attributes, `permanent` of type boolean that stores if the contractor is a permanent contractor or not and an `hourlyRate` attribute of type `double`.

   b) Write a constructor that takes all the arguments required to construct the `Person` class, as well as the two for this class.

   c) Add `setHourlyRate(double salary)` and `getHourlyRate()` methods.

4. Override the `toString()` method of the `Person` class in both of the sub-classes to print out the extra details created in each sub-class.

   Hint: Reuse existing functionality; do not duplicate code. Use the `super` keyword to call the `superclass` constructor as well as the superclass `toString()` method.

5. Write a `main()` method for each class to test the modifications. Create an instance, set the salary or rate as appropriate and then cause the `toString()` method to be invoked.

Exercise 4 – Polymorphism

This exercise will utilize the built-in functionality of object-oriented languages called polymorphism.

We will create a group of various types of Persons and print them out. The power of polymorphism will be demonstrated by observing that the appropriate `toString()` method is invoked, depending on the type of person (`Person`, `Employee` or `Contractor`).

Polymorphism works, because method resolution is not based on the compile-time class `supertype` declaration, but on the run-time class subtype.

1. Create a `PersonnelTest` class that holds an `array` of `Person` objects. Provide an `add()` method to add instances of Persons to the array. Also provide a `printPerson()` method that invokes the `toString()` method of each `Person` element in the array.

2. Provide a `main` method that creates instances of `Person`, `Employee` and `Contractor` objects, sets salary or hourly rate as appropriate, adds the instances to the `Person` array and calls the `printPersons()` method.

3. Check that the correct details are printed, based on subclass type.

Exercise 5 (Optional) – Implement Comparable

In production code, arrays frequently require sorting. The `Comparable` interface provides a way of defining how user-defined objects should be sorted. In this case, sort the array according to the person's age. The `Arrays` class provides sort functionality on an array argument.

Note: this exercise assumes that the `getAge()` method has been implemented in a previous exercise.

1. Have the `Person` class implement the `Comparable` interface.

2. In the `compareTo()` method, compare the age of the `Person` objects.

   Remember: The `compareTo()` method takes an object data type as an argument, therefore you will have to cast this object to a person.

3. In the `PersonnelTest` class, sort the `array` of `Person` objects by passing the array in to the `Arrays.sort()` method. This method will cause the overridden `compareTo()` method to be invoked.

4. Verify that the test output is now sorted by the person's age.

Exercise 6 – Create and Use an Interface

In a previous exercise, polymorphism was demonstrated with classes related by inheritance. The `Employee` and `Contractor` classes are subclasses of `Person`, inheriting common attributes and behaviors.

Employees and contractors also share a commonality in that they both get paid. However, getting paid is not a concept that all Persons share. This then can't be added to the `Person` class for `Employee` and `Contractor` to inherit. Since Java only allows for single inheritance, `Employee` and `Contractor` can't extend another class to share this functionality.

Java's answer to this is interfaces. Interfaces define abstract functionality that classes implement. Interfaces also provide polymorphic method resolution, but based on interface type rather than class-based inheritance. A class may implement one or more interfaces.

1. Create an interface named `Payable` that defines two methods named `calculateWeeklyPay()` and `getName()`.

2. Implement the interface in the `Employee` and `Contractor` classes. For `Employees`, weekly pay is calculated as salary divided by 52. For `Contractors`, weekly pay is calculated as `hourlyRate` times 40 (hours per week).

3. Note that the `getName()` method should already be defined in both classes. It is added to the interface to allow for the association of a name with a weekly pay amount. Because we will be accessing the instances by interface type, the class methods are not available unless they are part of the interface.

4. Create a test driver class named `PayableTest`. It will be very similar to `PersonnelTest`, but instead of holding an array of `Persons` it will hold an array of `Payables` and instead of a `printPersons()` method it will have a `printPaychecks()` method.

5. The `printPaychecks()` method should iterate though the array and invoke the `calculateWeeklyPay()` method on each `Payable`.

6. In the `main()` method, create several instances of `Employees` and `Contractors`, setting the salary or hourly rate as appropriate and add them to the array. Then invoke the `printPaychecks()` method.

7. Verify the weekly pay has been calculated correctly, based on whether the `Payable` is an `Employee` or a `Contractor`.