Lab 01 Report : Inheritance & Interfaces

Name: Jessica Lemon

Objective: To check your understanding of the properties of *Inheritance, Polymorphism, Abstract Classes and Interfaces*.

Topics	Exercises	Package
Inheritance	Employee Records	company

Part A: Employee Records

Overview:

Programs that are implemented on a large scale require that designers write clear, clean and versatile code. In this report, similar concepts are used to demonstrate the same effect on a smaller scale. Features such as loops, logic expressions for decision making, and object-oriented programming that has the ability to provide code reuse through inheritance, as well as changing behavior depending on object hierarchy. The level of coding redundancy and unnecessary code have also been reduced.

Code Updates, Functionality and Testing

Updates (Employee, Manager, Executive, and Client Program):

- The Employee class prints out the name, address, phone extension, salary and bimonthly pay. For this project, if an employee is a regular employee, they receive a salary of $100,000.00. This is considered the base salary for the company.

- Both Executive and Manager are *extensions* of the Employee class. If the employee is a manager, in addition to the base salary, they also receive experience pay, which is based on the number of years of experience multiplied by 12,000.00. If an employee is an executive, the receive four times the base salary and a monthly bonus. All employees are paid in 24 installments.

Functionality:

- *Employee (see details below)*: There are 3 constructors for various input usage. Accessors for pay and salary are populated here. However, they are expected to be overridden in the subclasses. Formatting here uses *String.format("String", data)* to see the console data more streamlined. Code re-use is utilized here, so that *toString* does not have to be copied in the subclasses of this class hierarchy. Also, able to compare records of the same type with the equals method (see *Keyword Description*).

- *Executive (see details below)*: There are 3 constructors for updating the Executive records. There is a method to update the executive bonus (*setBonus*) and to access new salary information. Also, able to compare records of the same type with the equals method (see *Keyword Description*).

- *Manager (see details below)*: There are 3 constructors for updating the Manager records. There is a method to update the years of experience for the manager (*setYears*) and to access new salary information. Also, able to compare records of the same type with the equals method (see *Keyword Description*).

Testing (Company Client Program):

- Test Records for Employee (“Jim Blake”), Manager (“Kim Mann”, “Tim Hill”) and Executive (“Sam Lowe”, “Tom Mitty”) were generated (hard coded) into program for testing. The program is modified to store the employees listed in an array (or an array list) and print out the list using a for-each loop.

- Printing to console is not the ideal testing resource to use, when testing larger data sets. Since this program is relatively small for the moment, printing statements will be sufficient. However, JUnit Tests will likely be implemented in future iterations of this program.

- The records for manager and executive will be tested using “Tim Hill” and “Tom Mitty”.

- The record for Tim Hill is modified to reflect the number of years of experience of 10 years. The record for Tom Mitty is modified to reflect the bonus to 8083.00.

- In the main method, each object “equals method” was used to compare each item to find the proper record and using their mutators adjusted the values for “experience” and “bonus”. Results were printed to console.

- All elements were checked for the relevant records, however to make this as general as possible each element was *cast* to either a manager or executive to allow a reduced decision-making process (see line 27-33 of CompanyClient code on GitHub).

Part B: Output

The following is the output generated from printing out the elements of the array data structure to console.

The General Output Displayed:

T E S T P R O G R A M

Name: Jim Blake
Address: Pike Street
Phone: 4022
Salary: 100000.0
Pay: 4166.7

Name: Tim Hill
Address: Pine Street
Phone: 3121
Salary: 160000.0
Pay: 6666.7

Name: Kim Mann
Address: High Street
Phone: 3315
Salary: 244000.0
Pay: 10166.7

Name: Sam Lowe
Address: Church Street
Phone: 2128
Salary: 460996.0
Pay: 19208.2

Name: Tom Mitty
Address: Market Street
Phone: 2124
Salary: 424996.0
Pay: 17708.2

Name: Tim Hill
Address: Pine Street
Phone: 3121
Salary: 220000.0
Pay: 9166.7

Issues:

- There is not an efficient way to distinguish between an executive and a manager other than the bonus and years of experience. It may be better served to create a label to identify these positions in future iterations.

- Some of the print out to console sometimes do not align well with other record outputs. May have to spend more time thinking about the formatting.

- Possible TODO's include, (i) running data from a file system to increase the number of records used f or testing. (ii) having line sections to separate records.

Testing Equals Method Output:

Code Test for Employee search compared with equal method.

T E S T P R O G R A M

Name: Tim Hill
Address: Pine Street
Phone: 3121
Salary: 220000.0
Pay: 9166.7

Name: Tom Mitty
Address: Market Street
Phone: 2124
Salary: 497002.0
Pay: 20708.4

Results:

Output was as expected, however more robust tests are need for a larger data set.

`Employee` Class

The `Employee` is a super class in the inheritance hierarchy and has the following specifications:

encapsulated data fields:

These design details will be encapsulated to provide enhanced security to the program. This will prevent random messages from altering the data in storage.

data field	description
`name`	stores an employee's name as a `String`.
`address`	stores an employee's address as a `String`.
`phone`	stores an employee's phone number as a `String`.

constructors:

These are the only constructors that will be used within the class construction. There seems to be no need to create a default constructor from the given design details since the record descriptions will always be required.

constructor	description
`public Employee (String name, String address, String phone)`	constructor that accepts three parameters: a name, address, and phone number.
`public Employee (Employee other)`	the `Employee` copy constructor.

methods:

The methods for this class are given “public” access to send messages back through method calls.

method	description	header
`equals`	returns `true`, if the contents of the current object is equal to the contents of another object, `false` otherwise. Uses the `instanceof` operator to regulate the objects under comparison.	`public boolean equals(Object obj)`
`pay`	returns the value of an employee’s annual pay divided into 24 payments.	`public double pay()`
`salary`	returns the base salary for an employee.	`public double salary()`
`toString`	returns the employee record in “pretty print” format.	`public String toString()`

`Executive` Class

The `Executive` class inherits the behavior from the `Employee` class and has the following specifications:

encapsulated data fields:

These design details will be encapsulated to provide enhanced security to the program. This will prevent random messages from altering the data in storage.

data field	description
`bonus`	stores the bonus of an executive employee's as a `double`.

constructors:

These are the only constructors that will be used within the class construction. There seems to be no need to create a default constructor from the given design details since the record descriptions will always be required.

constructor	description
`public Executive (String name, String address, String phone, double bonus)`	constructor that accepts four parameters: name, address, phone number and employee bonus.
`public Executive (Employee employee, double bonus)`	constructor that accepts two parameters: the employee record and employee bonus.
`public Executive (Executive other)`	the `Executive` copy constructor.

methods:

The methods for this class are given “public” access to send messages back through method calls.

method	description	header
`equals`	returns `true`, if the contents of the current object is equal to the contents of another object, `false` otherwise. Uses the `instanceof` operator to regulate the objects under comparison.	`public boolean equals(Object obj)`
`salary`	returns the salary for the current executive at bonus level.	`public double salary()`
`setBonus`	modifies/sets the employee bonus for executive	`public void setBonus(double bonus)`

`Manager` Class

The `Manager` class inherits the behavior from the `Employee` class and has the following specifications:

encapsulated data fields:

These design details will be encapsulated to provide enhanced security to the program. This will prevent random messages from altering the data in storage.

data field	description
`years`	stores the manager's years of experience as an `int`. Values are rounded up or down based on half year mark.

constructors:

These are the only constructors that will be used within the class construction. There seems to be no need to create a default constructor from the given design details since the record descriptions will always be required.

constructor	description
`public Manager (String name, String address, String phone, int years)`	constructor that accepts four parameters: name, address, phone number and employee bonus.
`public Manager (Employee employee, int years)`	constructor that accepts two parameters: the employee record and the employee's years of experience.
`public Manager (Manager other)`	the `Manager` copy constructor.

methods:

The methods for this class are given “public” access to send messages back through method calls.

method	description	header
`equals`	returns `true`, if the contents of the current object is equal to the contents of another object, `false` otherwise. Uses the `instanceof` operator to regulate the objects under comparison.	`public boolean equals(Object obj)`
`salary`	returns the salary for the current executive at bonus level.	`public double salary()`
`setYears`	modifies/sets the employee years of experience.	`public void setYears(int years)`

Keyword Descriptions

equals method:

The Object class contains methods that are common to all objects, such as the `equals` and `toString` methods.

Consider two `Employee` objects `e1` and `e2` that are allocated in different memory locations. The expression `e1 == e2` would evaluate to `false` because `e1` and `e2` do not refer to the same memory location. If we want to compare these objects based on their "state" (i.e. the contents of their data fields), we must use the `equals` method.

A class created **without** its own version of the `equals` method, receives a default version from the `Object` class. The `Object` class uses a very conservative version of equality, which considers two objects equal if they have the same memory address. This means that the default version behaves the same as `e1 == e2`.

If you want to make your `equals` method compare the contents (data fields) of two objects, you will have to create your own version of the `equals` method within your class definition file that replaces the default version of the `Object` class. A proper `equals` method performs a comparison of the two objects' contents and returns `true`, if all the objects' data fields are the same as the other object.

toString method:

Similarly, a class without a `toString` method also receives a default version of the `toString` method from the `Object` class. This default version would print for example something similar to Employee@116c082. To print the contents of the object properly, you must create a version in your class template that *overrides* (replaces) the default version in the `Object` class.

instanceof operator:

An operator called `instanceof` tests whether a variable refers to an object of a given type. This is useful to determine if there should be a cast to that type. **Note**: An `instanceof` test is a binary expression that produces a `boolean` result and is of the following form:

```
object instanceof Type
```

Example

```
if(object instanceof Employee){
     Employee other = (Employee) object;
}
```