CSC-240 Lesson 2: Classes and Objects

Assignment 2

Overview

The following assignment is *based* on the Chapter 3 Exercise 3.13 (Employee Class), but asks for some additional features. All the needed information is contained in this document (topic).

Submit just one file for this assignment.

- `Employee.java`—file that defines the **Employee** class.

A test program for this class is already written for you. The test program is supplied (see below). It creates instances of the class that you will write, and checks as far as is possible that the class satisfies the requirements presented in the next sections.

Provided as well is an *outline* or *skeleton* of the **Employee** class matching the UML description.

The test program and the skeleton may be downloaded by using this link to the zipfile containing the source code for the **EmployeeTest** class and the **Employee** class outline (see Postscript). More information about using these resources is included in the descriptions below. The test program is the main class (so you do not need to provide a *main* method), and it exercises the constructors and methods of the **Employee** class, checking that the right things happen. Of course, like all testing, not everything required in the **Employee** class can be checked in this manner, so the Instructor will also read the code that has been written, looking for both good programming practices, and for correct actions or structure that cannot be checked programmatically.

Implementing Employee

In this exercise you will write the code to implement the class named **Employee**.

[You do not have to write a test program: a test program that you may download is provided. This test program creates **Employee** objects as described in the textbook, and does the needed actions to increase the salaries. The test program also does some verification of other features.]

The UML below defines the needed field variables (three of them) and their types, the constructor, and the needed methods.

- Create the three instance variables as requested in the textbook. The names for these instance variables are specified in the UML below.
- Define a constructor for the class. The details of the constructor interface are shown in the UML specification below. The values supplied as arguments for the constructor are used to initialize the corresponding fields.
- Create the getters and setters for each of the properties *first*, *last* and *monthlySalary*, that will use these instance variables. (Note the names of the properties are not the same as the identifiers of the instance variables.)
- Create a method in the **Employee** class, named *displayValues*, that writes to the standard output file the employee's ***complete*** name, ***monthly*** salary, and ***annual*** salary. The test program will use calls to that method to display **Employee** object values.
- Override the inherited *toString* method (the format of the output of the *toString* method is described below).
- Write code where needed to report errors found in calls made to the constructor or methods of the class. These error messages should be written to the standard output file. (In the sample below, the **Employee** class has preceded all its printed output with the word “Employee.” The test program—which you will use to evaluate your implementation of the **Employee** class—precedes all its output with “**”, or “**??” (if the message indicates an error that should be corrected.)

The following UML provides a description of the API for the class. In this description, “-” indicates a **private** member, and “+” indicates a **public** constructor or member:

Employee

- firstNameField: String
- lastNameField: String
- salaryField: double

+<constructor> Employee(firstName: String, lastName: String, initialSalary: double)
+ displayValues()
+ getFirst(): String
+ setFirst(firstName: String)
+ getMonthlySalary(): double
+ setMonthlySalary(salary: double)
+ getLast(): String
+ setLast(lastName: String)
+ toString(): String

(It is not usual to specify in advance **private** members in UML, but is done here to ensure that the class submitted is gradeable by the Instructor. However, when UML is used to describe a class that has been implemented, **private** members will often be shown so that the internal operation of the class can be described for the benefit of later programmers and maintainers of the code.)

In the zipfile mentioned above, there is a Java source code outline (or "skeleton") class for the class **Employee**. This outline contains essentially the same information as the UML description above, except that it is in Java. By "skeleton" is meant that all the required methods and fields are present, but they do not yet contain correct values or code (just as a human skeleton defines the basic shape of a body, but does not have any flesh or muscles). The advantage of this skeleton over the UML is that it can be used directly as a starting point for creating the needed implementation of the class—to implement your version of the class, you may simply modifiy the skeleton. As provided, it does allow the test program to run, but, of course, lots of error messages will be seen.

Defining the behavior of the Employee class

The definitions supplied above describe just the form and syntax of the API. They only hint at the actual function of the constructor and methods. (Note that for the purposes of testing, the "user" in the following descriptions is the test program.) The following defines the desired behaviors:

- The monthly salary value supplied must not be not negative. If the salary supplied by the user of the class is negative, that must be reported as a message on the standard output. If the salary given is invalid (*i.e.*, is negative), the salary for the **Employee** should be set to zero. If the value supplied was invalid, the getter will return the zero value that was stored, otherwise the supplied value. (See examples of invalid salary reporting below.)

- The **String**s supplied for the first and last names of the employee must not be **null** nor empty. If a name supplied by the user of the class is **null** or empty, that must be reported as a message on the standard output. If a name given is invalid (*i.e.*, is **null** or empty) the corresponding field, *firstNameField* or *lastNameField*, should be set to the value **null**. If a name supplied was invalid, the corresponding getter will return that **null** value, otherwise the supplied value. (See examples of invalid name reporting below.)

- The *displayValues* method will show the **String** "<Invalid first name>" or "<Invalid last name>" whenever the corresponding name as supplied by the user of the class was invalid. (See examples of "display values" output below.) Note that the output produced by *displayValues* should end with an appropriate line separator.

- The *toString* method must produce a **String** that starts with the value that is the value of the *toString* method of the superclass (which is by definition **Object**) of the **Employee** instance, and then ***concatenates*** a space character, the first name, another space character, and the last name (as returned by the corresponding getters). Note that the returned *toString* value does ***not*** end with a line separator. (This is to be a debugging-style *toString* method, so no checks need to be made on any of the values included in the resultant **String**. Please read the article "FAQ How to override the method toString" in the *Informational Resources* module for more specific discussion.)

The **Employee** class skeleton outline fulfills the requirements of the UML-defined interface (it has the correct names and types), but does not contain the correct implementations. Replacing the *pro forma* code (that simply enables the class to compile) with the semantically correct code is the task for this assignment.

Testing the Employee Class

The exercise as stated in the textbook asked that you also write a test program. However, this has already been done, and the test program is also contained in the zipfile, and has the name **EmployeeTest**.

There is no need to, and you should not, attempt to modify the **EmployeeTest** class.

The **EmployeeTest** class is a main program. You should not write a *main* method in your **Employee** class—the test program contains the needed *main* method. Just run the **EmployeeTest** class as the first called class of the program.

The **EmployeeTest** class does everything requested of the test program as described in the textbook, but also checks some other functions of the **Employee** class implementation. Example output from the test program is shown below. The test program tests values as directed by the problem in the textbook. This is a very minimal test program: in real life it would be a good idea to be even more thorough and to make even more tests. Nevertheless, like many test programs, the test program is (already) longer and more complex that the class that it is testing.

The example output below is the result of running the test program with a completed implementation of the **Employee** class, so when you complete your version of the **Employee** class, the output when running with the test program should be similar.

In this example output, all the lines indented and starting with "**" are lines created by the test program. Your **Employee** class should generate messages similar to all the other lines (follow the code through the test program to see what is happening). If you, when running your version, see lines printed by the test program indented and starting with "**??", they are messages indicating an error that the test program has found in the implementation of the class **Employee** (an error that you should be able to correct).

In this particular example, the output with the lines beginning with "Employee:" are generated by the **Employee** class, either as output from the *displayValues* method, or are produced as error messages (these latter also including the word "Error"). You do not have to reproduce these messages from the **Employee** class exactly—the explanation here is simply to make it easier for you to understand what messages come from the test program (*i.e.*, those with "**"), and those that come from the class being tested.

```
**CSC-240 Assignment 2

    ** --- TESTING CREATING VALID Employee OBJECTS ---

    **Creating two valid employees
    **Checking the nominal employee Bob Jones
    **Bob Jones is OK
    **The result of "displayValues" for that employee
Employee: Bob Jones; Monthly Salary: 2985.00; Annual Salary: 35820.00
    **Checking the nominal employee Susan Bakker
    **Susan Bakker is OK
    **The result of "displayValues" for that employee
Employee: Susan Bakker; Monthly Salary: 3361.75; Annual Salary: 40341.00
    **Changing first name of Bob Jones to Robert
    **Checking the nominal employee Robert Jones
    **Robert Jones is OK
    **The result of "displayValues" for that employee
Employee: Robert Jones; Monthly Salary: 2985.00; Annual Salary: 35820.00
    **Changing last name of Susan Bakker to Baker
    **Checking the nominal employee Susan Baker
    **Susan Baker is OK
    **The result of "displayValues" for that employee
Employee: Susan Baker; Monthly Salary: 3361.75; Annual Salary: 40341.00

    ** --- INCREASING Employee SALARIES BY 10% ---

    **Checking the nominal employee Robert Jones
    **Robert Jones is OK
    **The result of "displayValues" for that employee
Employee: Robert Jones; Monthly Salary: 3283.50; Annual Salary: 39402.00
    **Checking the nominal employee Susan Baker
    **Susan Baker is OK
    **The result of "displayValues" for that employee
```

```
Employee: Susan Baker; Monthly Salary: 3697.93; Annual Salary: 44375.10

    ** --- TESTING INVALID CALLS TO Employee METHODS ---

    **Setting last name of Robert Jones to ""
Employee: Error in setting object: last name is a null or empty String
    **Checking the nominal employee Robert <null>
    **Robert <null> is OK
    **The result of "displayValues" for that employee
Employee: Robert <Invalid last name>; Monthly Salary: 3283.50; Annual Salary: 39402.00
    **Setting first name of Robert Jones to null
Employee: Error in setting object: first name is a null or empty String
    **Checking the nominal employee <null> <null>
    **<null> <null> is OK
    **The result of "displayValues" for that employee
Employee: <Invalid first name> <Invalid last name>; Monthly Salary: 3283.50; Annual Salary: 39402.00
    **Setting first name of Susan Baker to ""
Employee: Error in setting object: first name is a null or empty String
    **Checking the nominal employee <null> Baker
    **<null> Baker is OK
    **The result of "displayValues" for that employee
Employee: <Invalid first name> Baker; Monthly Salary: 3697.93; Annual Salary: 44375.10
    **Setting last name of Susan Baker to null
Employee: Error in setting object: last name is a null or empty String
    **Checking the nominal employee <null> <null>
    **<null> <null> is OK
    **The result of "displayValues" for that employee
Employee: <Invalid first name> <Invalid last name>; Monthly Salary: 3697.93; Annual Salary: 44375.10
    **Setting salary of Susan Baker to -1000.0Employee: Error in setting object: Salary is less than zero: -1000.00
    **Checking the nominal employee <null> <null>
    **<null> <null> is OK
    **The result of "displayValues" for that employee
Employee: <Invalid first name> <Invalid last name>; Monthly Salary: 0.00; Annual Salary: 0.00
    **Creating "" Smith, with salary of -500.0
Employee: Error in setting object: first name is a null or empty String
Employee: Error in setting object: Salary is less than zero: -500.00
    **Checking the nominal employee <null> Smith
    **<null> Smith is OK
    **The result of "displayValues" for that employee
Employee: <Invalid first name> Smith; Monthly Salary: 0.00; Annual Salary: 0.00
    **Creating <null> Baker
Employee: Error in setting object: first name is a null or empty String
    **Checking the nominal employee <null> Baker
    **<null> Baker is OK
    **The result of "displayValues" for that employee
Employee: <Invalid first name> Baker; Monthly Salary: 500.00; Annual Salary: 6000.00
```

Postscript

The easiest way in NetBeans to set up the **EmployeeTest** class and the **Employee** class in the <default package> is firstly to unpack the zipfile. This will create the two files, `EmployeeTest.java` and the `Employee.java`. Now, in NetBeans, create a project for the assignment in the usual way (but ***not*** checking the box “Create Main Class.”) The project created will be empty. Now, select each of the files (or both simultaneously) and drag them onto the “Source Packages” icon in the project. The result will be the project now containing the classes **EmployeeTest** and **Employee** in the <default package>. The **Employee** class skeleton may now be modified to become the file that you submit for this assignment. Just remember to add your name and student ID as a new `@author` line in the class-level comment.

More detailed instructions on getting files from a zipfile into a NetBeans project may be found in the article “FAQ How do I get Samples and Demonstrations into NetBeans” in the *Informational Resources* module.

The following *About the Assignment* topic contains further explanation, directions, and hints about this assignment. Please read that topic before starting this assignment.

Important

Please follow the directions of Programming Assignment Identification in submitting your programming solutions.

If you have any questions or concerns about the Assignment or the *About the Assignment* topic, please use the Lesson Question discussion topic, or send a message by the D2L Internal Messaging system if you prefer.

Note: you may submit to the Assignment Submission Folders as many times and as often as you wish, up to the deadline time. Each submission is tagged with the date/time, and so each submission remains separate and distinct. Unless you leave instructions to the contrary, only the most recent of each file with the same name will be viewed for the purposes of grading. Details may be found in the topic *How To Submit and Get Feedback on Assignments* in the *How To* module.

Messages that accompany Assignment Submissions are read, and responded to, ***only*** when assignment submissions are graded (which is after the Assignment Submission Folder closing date/time). If you have a comment or question about an assignment, or a request for assistance, that needs an earlier response, then that comment, question or request should be made or asked *via* an Internal Message or the Discussion board, as these are usually read and answered every day.