Department of Computer Science
University of Pretoria

Programming Languages
COS 333

Practical 7:

Object-Oriented Programming

May 20, 2024

1 Objectives

This practical aims to achieve the following general learning objectives:

- To gain and consolidate some experience writing object-oriented programs in Ruby;
- To consolidate a variety of basic concepts related to object-oriented programming languages, as presented in the prescribed textbook for this course.

2 Plagiarism Policy

Plagiarism is a serious form of academic misconduct. It involves both appropriating someone else's work and passing it off as one's own work afterwards. Thus, you commit plagiarism when you present someone else's written or creative work (words, images, ideas, opinions, discoveries, artwork, music, recordings, computer-generated work, etc.) as your own. Note that using material produced in whole or part by an AI-based tool (such as ChatGPT) also constitutes plagiarism. Only hand in your own original work. Indicate precisely and accurately when you have used information provided by someone else. Referencing must be done in accordance with a recognised system. Indicate whether you have downloaded information from the Internet. For more details, visit the library's website: `http://www.library.up.ac.za/plagiarism/`.

3 Submission Instructions

Upload your practical-related source code file to the appropriate assignment upload slot on the ClickUP course page. You must implement and submit your entire Ruby program in a single source file named `s99999999.rb`, where `99999999` is your student number. Multiple uploads are allowed, but only the last one will be marked. The submission deadline is **Monday, 27 May 2024, at 12:00**.

4 Background Information

For this practical, you will be writing a program in Ruby 2.5. The course website contains documentation related to Ruby [1]. You will be implementing all classes in a single source file.

In order to complete this practical, you will have to research various concepts in Ruby, which are relatively different to other object-oriented programming languages you will be used to. In particular, you will have to find out how constructors work in Ruby. You will also need to research accessor methods (attribute readers and writers), which are shorthand approaches for writing getters and setters. Additionally, you will also have to investigate lists in Ruby.

5 Practical Task

This practical requires you to build a simple system that represents a business containing departments with personnel who can be managers or employees, and computes salary information. This implementation must use object-oriented programming concepts. The following details must be included in your implementation:

An `Employee` class. This class is not meant to be instantiated. The class should have the following characteristics:

- Three instance variables: `earnings`, `baseSalary`, and `baseSalaryPaid`. The `earnings` variable is a real value representing the total earnings of an employee over the time they work at the company. The `baseSalary` is a real value representing the base salary an employee earns every month. An employee can only earn their base salary once per month, so the `baseSalaryPaid` is a Boolean variable that represents whether the employee has already earned their base salary this month.
- A constructor that receives one parameter and uses this value to set the base salary, while also setting the `earnings` to zero, and the `baseSalaryPaid` to false. Ruby only supports a single constructor for a class, but there should be a way to provide no parameters to the constructor, which sets the base salary value to a default of 20 000.00.
- An attribute reader accessor method for the `earnings` instance variable. This returns the earnings of the employee.
- An instance method, called `payEmployee`, which pays the employee their base salary at the end of a month. The method should first check whether the base salary has already been paid for the month (using the `baseSalaryPaid` instance variable). If the base salary has not been paid, the method should simply increment `earnings` by the base salary for the employee. If the base salary has been paid, no action should be taken.
- A method, called `newMonth`, which simply sets `baseSalaryPaid` to false.

A derived classes of `Employee`, called `Manager`. Instances of this class can be created. The class should have the following characteristics:

- In addition to all the instance variables inherited from the `Employee` class, each `Manager` has an additional bonus that varies from manager to manager. The bonus is paid for each team the manager belongs to (see the description for the `Team` class, below).
- A constructor that receives one parameter, and uses it to set the bonus. The constructor also calls the parent class constructor, in order to set the default base salary and initialise all the other inherited instance variables.
- The class overrides the `payEmployee` method inherited from the parent class. This method ensures the base salary is paid (by calling the `payEmployee` method in the parent class), and also increments the salary by the bonus for the manager. Note that there is no condition on the bonus being paid. This means that the first time `payEmployee` is called, the base salary of the manager is paid, as well as their bonus. All subsequent calls to `payEmployee` will only result in the bonus being paid.

Another derived class of `Employee`, called `Programmer`. Instances of this class can be created. The class should have the following characteristics:

- No additional instance variables are provided by the `Programmer` class.
- A constructor that receives a real valued percentage as a parameter (where 0.5 indicates 50% and 1.0 represents 100%). The constructor computes the provided percentage of 20 000.00 as the computed salary of the programmer, and then calls the parent class constructor with the computed salary as a parameter.

A class called `Team`. The class should have the following characteristics:

- One instance variable, which stores a a list of employees. Additionally, you must provide:
- An instance method, named `addMember`, which adds an individual employee (specified by a parameter) to a team. Allow only a maximum of two employees to be added, and have the method throw an exception if an attempt is made to add a third employee.
- An instance method called `payTeam`, which uses the `payEmployee` method in the `Employee` class to pay the employees of the team. Note that this will result in each manager being paid their base salary once, with a bonus added for each team that they manage. In contrast, each programmer will only be paid their base salary, regardless of how many teams they are a member of.
- An instance method called `newMonth`, which calls the `newMonth` method for the employees of the team.
- An instance method called `printEarnings`, which prints the earnings for the manager and each team member in the list. Use the attribute reader for the earnings of each employee.

Finally, write test code that creates two teams. Prompt the user for the details required to create a single manager. Add this manager to both teams. Then, for each team, prompt the user for the details required to create one programmer, and add this programmer to the team in question. Once all the team members have been created and added, prompt the user the enter a "y" if they want to pay the teams for another month of work (this requires paying the team, and starting a new month), and "n" to end the payments. Finally, once the user has decided to make no further payments, print the earnings of each team member. This final output should demonstrate that all employees have been paid correctly, according to the above specification (in particular, because the manager is on two teams, they should only be paid their base salary once, but should receive two bonuses, per month).

Be sure to utilise as much functionality from parent classes as possible, and do not re-implement functionality that exists in parent classes.

6 Marking

Submit the Ruby implementation to the appropriate assignment upload slot. Do not upload any additional files other than your source code. Both the implementation and the correct execution of the program will be taken into account. Your program code will be assessed during the practical session in the week of **Monday, 27 May 2024**.

References

[1] Huw Collingbourne. The book of Ruby. 2009.