OCaml

Object Oriented Programming - 1

Summary: The main theme of this module is to introduce the object oriented programming style with OCaml.

Version: 1.00

Contents

Chapter I

Foreword: The Ultimate Mega Gangsta Chocolate Butter Cake

I.1 Ingredients:

- 250g of chocolate
- 250g of butter (Yeah... I know)
- 150g of sugar
- 4 eggs
- 1 large spoonful of flour

I.2 Recipe:

- Melt the chocolate with the butter.
- Add the sugar.
- Add the eggs one at a time, mixing well between each addition.
- Gradually add the flour, mixing it into the batter.
- Pour the batter into a cake pan (or something similar).
- Place the cake pan into a larger pan with a little water in it (bain-marie style).
- Bake both pans together in the oven for 45 minutes at 180°C.
- Refrigerate for 12 hours (Not a joke).

Done!

Chapter II

General rules

- Your project must be realized in a virtual machine.
- Your virtual machine must have all the necessary software to complete your project. These softwares must be configured and installed.
- You can choose the operating system to use for your virtual machine.
- You must be able to use your virtual machine from a cluster computer.
- You must use a shared folder between your virtual machine and your host machine.
- During your evaluations you will use this folder to share with your repository.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

Ocaml piscine, general rules

- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names, etc.
- Unless otherwise explicitly stated, the keywords `open`, `for` and `while` are forbidden. Their use will be flagged as cheating, no questions asked.
- Turn-in directories are `ex00/`, `ex01/`, . . . , `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `OCaml` syntaxes you learned about since the beginning of the piscine, you are not allowed to use any additional syntaxes, modules and libraries unless explicitly stated otherwise.
- The exercices must be done in order. The graduation will stop at the first failed exercice. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `ocamlopt`. When you are required to turn in a function, you must also include anything necessary to compile a full executable. That executable should display some tests that prove that you've done the exercise correctly.
- Remember that the special token `";;"` is only used to end an expression in the interpreter. Thus, it must never appear in any file you turn in. Regardless, the interpreter is a powerfull ally, learn to use it at its best as soon as possible!
- The subject can be modified up to 4 hours before the final turn-in time.
- In case you're wondering, no coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code he or she can't grade. As usual, big functions are a weak style.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the

exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.

- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter IV

Day-specific rules

- This day is all about fun. Bonus points will be awarded if you include clever references to the Doctor Who universe.
- You are in a functional programming "piscine," so your coding style MUST be functional (except for input/output side effects). I insist: your code MUST be functional; otherwise, your defence session could be tedious.
- For each exercise, you must provide sufficient material for testing during the defence session. **Any functionality that cannot be tested will not be graded!**
- Don't slack off. If something in an exercise is not tested, the defence will end immediately.

Chapter V

Exercise 00: Do What I do. Hold tight and pretend it's a plan!

	Exercise 00
Exercise 00: Do What I do. Hold tight and pretend it's a plan!	
Turn-in directory : *ex*00/	
Files to turn in : `people.ml, main.ml, Makefile`	
Allowed functions : `Stdlib modules`	

- Write a class `people` that has the following attributes:
  - A `name` attribute of type `string`.
  - An `hp` attribute of type `int`, initialized to 100.
  - A `to_string` method that returns a string representing the object's name and attribute values.
  - A `talk` method that prints the following string to the standard output:
    I'm [NAME]! Do you know the Doctor?
  - A `die` method that prints the following sentence to the standard output: *Aaaarghh!*
  - An initializer message that indicates the object has been created (feel free to make this message creative or descriptive!).
- You must simulate all the methods in the `main.ml` file to provide sufficient testing for the defence.

Chapter VI

Exercise 01: The Name of the Doctor!

	Exercise 01
Exercise 01: The Name of the Doctor!	
Turn-in directory : *ex*01/	
Files to turn in : `doctor.ml, people.ml, main.ml, Makefile`	
Allowed functions : `Stdlib modules`	

- Write a class *doctor* that has the following attributes:
  - A ‘name‘ attribute of type ‘string‘.
  - An ‘age‘ attribute of type ‘int‘.
  - A ‘sidekick‘ attribute of type ‘people‘.
  - An ‘hp‘ attribute of type ‘int‘ initialized to 100.
  - A ‘to_string‘ method that returns the name of the object along with the values of its attributes.
  - A ‘talk‘ method that prints the following string to the standard output:
    Hi! I’m the Doctor!
  - An initializer which indicates that the object has been created (feel free to use something explicit and descriptive to announce it!).
  - A ‘travel_in_time‘ method that takes two arguments of type ‘int‘: *start* and *arrival*, and changes the Doctor’s age logically (Think before coding any unusual arithmetic... Please...). This method also draws a TARDIS on the standard output. (If you don’t know what a TARDIS is, google it!)
  - A ‘use_sonic_screwdriver‘ method that prints the following sentence to the standard output: *Whiiiiwhiiiwhiii Whiiiiwhiiiwhiii Whiiiiwhiiiwhiii*
  - A private ‘regenerate‘ method that sets the Doctor’s ‘hp‘ to 100 (the maximum).

- You must simulate all the methods in the main to provide sufficient testing for the evaluation.

Chapter VII

Exercise 02: You are a good Daaaaaalek!

Exercise 02
Exercise 02: You are a good Daaaaaalek!
Turn-in directory : *ex*02/
Files to turn in : `dalek.ml, doctor.ml, people.ml, main.ml, Makefile`
Allowed functions : `Pervasives, String and Random modules`

- Write a class *dalek* that has the following attributes:
    - A ‘name‘ attribute of type ‘string‘ randomly generated with the format: *DalekXXX*, where XXX is a random set of characters (e.g., DalekSec).
    - An ‘hp‘ attribute of type ‘int‘, initialized to 100.
    - A ‘shield‘ attribute of type ‘bool‘, mutable, initialized to ‘true‘ and changes its value each time the ‘exterminate‘ method is used.
    - A ‘to_string‘ method that returns the name of the object along with attribute values.
    - A ‘talk‘ method that randomly prints one of the following strings to the standard output:
        - *Explain! Explain!*
        - *Exterminate! Exterminate!*
        - *I obey!*
        - *You are the Doctor! You are the enemy of the Daleks!*
    - An ‘exterminate‘ method that takes a ‘people‘ object as an argument and kills it instantly.
    - A ‘die‘ method that prints the following sentence to the standard output: *Emergency Temporal Shift!*

- Simulate a battle between the Doctor, a Dalek, and a human in the main to provide sufficient testing for the evaluation. Feel free to add any setters you need.

Chapter VIII

Exercise 03: The Day of The Doctor!

	Exercise 03
Exercise 03: The Day of The Doctor!	
Turn-in directory : *ex*03/	
Files to turn in : `army.ml, dalek.ml, doctor.ml, people.ml, main.ml, Makefile`	
Allowed functions : `Stdlib and List modules`	

- Write a parameterized class *army* that has the following attributes:
  - A 'members' attribute of type ''a list' which contains a list of instances of one of the three previous classes.
  - An 'add' method that adds an instance to the list (either to the front or back).
  - A 'delete' method that removes the head of the 'members' list (either from the front or back).
- Simulate the construction and destruction of an army of each type in the main to provide sufficient testing for the evaluation.

`This exercise is not mandatory.`

Chapter IX

Exercise 04: The Time War!

	Exercise 04
Exercise 04: The Time War!	
Turn-in directory : *ex*04/	
Files to turn in : `galifrey.ml, army.ml, dalek.ml, doctor.ml, people.ml, main.ml, Makefile`	
Allowed functions : `Everything functional (so no while, for, Array, etc.)`	

- Write a class *galifrey* with the following attributes:
    - A ‘members‘ attribute of type ‘dalek list‘, containing instances of the ‘dalek‘ class.
    - A ‘members‘ attribute of type ‘doctor list‘, containing instances of the ‘doctor‘ class.
    - A ‘members‘ attribute of type ‘people list‘, containing instances of the ‘people‘ class.
    - A ‘do_time_war‘ method that launches the greatest battle across time and space. You will need to add more methods to handle proper behavior. For example, you might need one method to select an attack for an instance, or

another to check if any object in a list is still alive. Feel free to add any method that seems necessary.

- Simulate a time war in the main to provide sufficient testing for the evaluation.

`This exercise is not mandatory.`

Chapter X

Submission and Peer Evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your folders and files to ensure they are correct.

`The evaluation process will take place on the computer of the evaluated group.`