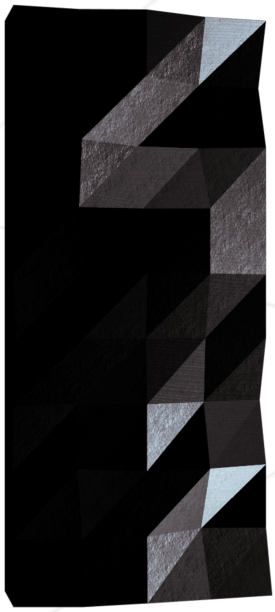

Ocaml

OCaml's modules language - 1

Staff 42 bocal@staff.42.fr

Summary: The main theme of this day is related to the ocaml modules.

Version: 1.00

Contents

Chapter I

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 II

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 III

Exercise 00: Cards colors

	Exercise 00
Exercise 00: Cards colors	
Turn-in directory : $ex00/$	
Files to turn in : `Color.ml and main.ml`	
Allowed functions : `Nothing`	

Regular play cards fit nicely as a programming topic when dealing with modules and nested modules. Colors, values, cards and decks, all tied together in a smart design.

As a start, we need to represent cards colors, namely `spade`, `heart`, `diamond` and `club`, as an `OCaml` type and instrument that type with relevant values and functions.

Write the file `Color.ml` that respects the following interface:

```
type t = Spade | Heart | Diamond | Club

val all : t list                        (** The list of all values of type t *)

val toString        : t -> string                   (** "S", "H", "D" or "C"  *)
val toStringVerbose : t -> string                   (** "Spade", "Heart", etc *)
```

Provide some tests in the file `main.ml` to prove that your `Color` module works as intended.

Chapter IV

Exercise 01: Cards values

	Exercise 01
Exercise 01: Cards values	
Turn-in directory : $ex01/$	
Files to turn in : `Value.ml and main.ml`	
Allowed functions : `invalid_arg`	

We have colors, now we need values for our cards. Cards values form a total ordered set, we need a type to represent them, and values and functions to instrument that type. The card values of a regular 52 cards deck are 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king and as.

Write the file `Value.ml` that respects the following interface:

```
type t = T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | Jack | Queen | King | As

(** The list of all values of type t                                    *)
val all : t list

(** Interger representation of a card value, from 1 for T2 to 13 for As  *)
val toInt           : t -> int

(** returns "2", ..., "10", "J", "Q", "K" or "A"                        *)
val toString        : t -> string
(** returns "2", ..., "10", "Jack", "Queen", "King" or "As"             *)
val toStringVerbose : t -> string

(** Returns the next value, or calls invalid_arg if argument is As      *)
val next      : t -> t
(** Returns the previous value, or calls invalid_arg if argument is T2  *)
val previous  : t -> t
```

Provide some tests in the file `main.ml` to prove that your `Value` module works as intended.

Chapter V

Exercise 02: Cards

	Exercise 02
Exercise 02: Cards	
Turn-in directory : $ex02/$	
Files to turn in : `Card.ml and main.ml`	
Allowed functions : `invalid_arg, Printf.sprintf and the List module`	

We have colors and values, now we can have cards ! Write the file `Card.ml` that respects the interface below. Several things to note regarding this interface:

- The `Card` module embeds the `Color` and `Value` modules. Just copy your previous code in the corresponding structures.
- The type `Card.t` is abstract. That means you're free to implement it as you want. Choose wisely, some solutions are better than otters. And otters are cute.
- All values' and functions' types and identifiers are self explainatory. Just read and use your brain, no tricks here.
- The function `toString : t -> string` returns strings like: `"2S"`, `"10H"`, `"KD"`, ...
- The function `toStringVerbose : t -> string` returns strings like: `"Card(7, Diamond)"`, `"Card(Jack, Club)"`, `"Card(As, Spade)"`, ...
- The function `compare : t -> t -> int` behaves like the Pervasives `compare` function.
- The functions `max` and `min` return the first parameter if the two cards are equal.
- The function `best : t list -> t` calls `invalid_arg` if the list is empty. If two or more cards are equal in value, return the first one. True coders use `List.fold_left` to do this function.

Provide some tests in the file `main.ml` to prove that your `Card`, `Card.Color` and `Card.Value` modules work as intended.

```
module Color :
sig

    type t = Spade | Heart | Diamond | Club

    val all : t list

    val toString        : t -> string
    val toStringVerbose : t -> string

end


module Value :
sig

    type t = T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | Jack | Queen | King | As

    val all : t list

    val toInt           : t -> int
    val toString        : t -> string
    val toStringVerbose : t -> string

    val next      : t -> t
    val previous  : t -> t

end


type t

val newCard : Value.t -> Color.t -> t

val allSpades   : t list
val allHearts   : t list
val allDiamonds : t list
val allClubs    : t list
val all         : t list

val getValue : t -> Value.t
val getColor : t -> Color.t

val toString        : t -> string
val toStringVerbose : t -> string

val compare : t -> t -> int
val max     : t -> t -> t
val min     : t -> t -> t
val best    : t list -> t

val isOf      : t -> Color.t -> bool
val isSpade   : t -> bool
val isHeart   : t -> bool
val isDiamond : t -> bool
val isClub    : t -> bool
```

Chapter VI

Exercise 03: Deck

	Exercise 03
Exercise 03: Deck	
Turn-in directory : $ex03/$	
Files to turn in : `Deck.mli, Deck.ml and main.ml`	
Allowed functions : `Allowed functions and modules from the previous exercices, plus raise and the Random module`	

We have cards, it's time to organize them in a deck represented by the `Deck` module. First write the interface for that module in the file `Deck.mli` according to the following statements:

- The `Deck` module embeds the `Card` module from the previous exercice.
- The `Deck` module exposes an **abstract** type `t` that represents a deck. Its definition is up to you.
- The `Deck` module exposes a function `newDeck` that takes no argument and returns a deck of the 52 cards (i.e. the type `t`) in **random** order. This means that upon two different calls to the function `newDeck`, the order of the deck will be different.
- The `Deck` module exposes a function `toStringList` that takes a deck as a parameter and returns a list of the string representations of each card.
- The `Deck` module exposes a function `toStringListVerbose` that takes a deck as a parameter and returns a list of the verbose string representations of each card.
- The `Deck` module exposes a function `drawCard` that takes a deck as a parameter and returns a couple composed of the first card of the deck and the rest of the deck. If the deck is empty, raise the exception `Failure` with a relevant error message.

Now implement the `Deck` module in the file `Deck.ml` according to its interface.

Provide some tests in the file `main.ml` to prove that your `Deck`, `Deck.Card`, `Deck.Card.Color` and `Deck.Card.Value` modules work as intended.

`This exercise is not mandatory.`

Chapter VII

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 happen on the computer of the evaluated group.`