CISS445 Lecture 16: Modules

Yihsiang Liow

January 29, 2020

Table of contents I

1 Namespace

2 Abstraction and signature

Namespace I

- Many programming languages provide the concept of modules.
- You can think of a module as a standard alone library.
- A module provides
  - Namespace
  - Abstaction

  for names, functions, classes, etc.
- Modules are helpful managing large-scale projects by breaking up software into smaller pieces which can be written and tested separately.
- Modules also encourage code reuse.
- The simplest use is to act as a namespace, i.e., a named container for a collection of names.

C++ namespace I

- In C++

```
// File: X.h
namespace X
{
    extern int x;
};
```

```
// File: X.cpp
namespace X
{
    int x = 42;
};
```

C++ namespace II

```
// File: main.cpp
#include <iostream>
#include "X.h"
int main()
{
    std::cout << X.x << '\n';
    return 0;
}
```

Python I

- In python3, a module is a python file.

```
// File: X.py
x = 42
```

```
// File: main.py
import X
print(X.x)
```

OCAML I

- ```
  (* File: X.ml *)
  module X =
    struct

      let x = 42;;

    end
  ;;
  ```

```
(* File: main.ml *)
#use "X.ml";;
Printf.printf "%d\n" X.x;;
```

- If you load `X.ml` in utop, you'll see

```
module X : sig val x : int end
```
```

OCAML II

- In the above example, the module can also be placed in `main.ml`.
- In OCAML a module need not be a file, i.e., an OCAML file can contains many namespaces (like C++).
- Exercise. Create two different OCAML modules and put `x` in both and bind them to values of different types. Print these two `x`'s from the two modules.
- Exercise. Create a module `A` that contains
  - the name `x` bound to 42 and
  - another module `B` which contains name `x` bound to `3.14`

  Print the two `x`'s. This tells you that OCAML modules can be nested.

OCAML III

- Exercise. Write a `Stack` module in a file named `stack.ml`. This implements a stack of integers using an int list. Operations include `push`, `pop`, `is_empty`, `peek`, `print`.

Signature I

- Load this in utop:

```
module type M =
  sig
    val f: int -> int;;
  end
;;
module M0: M =
  struct
    let f x = x + 1;;
  end
;;
module M1: M =
  struct
    let f x = x + 2;;
```

Signature II

```
  end
;;
```

- Next, apply `M0.f` to 5.
- Next, apply `M1.f` to 5.
- The structure `M0` and `M1` matches the signature of `M`.
- `M` is like a C++ abstract base class while `M0`, `M1` are like concrete subclasses.
- Next, type this in utop:

```
module M2: M =
  struct
    let f x = x +. 1.0;;
  end
;;
```

Signature III

There's a signature mismatch. Read the error message.

- Frequently a module implements a data structure (list, stack, queue, tree, etc.).
- Here's the abstract definition (signature) of stack

```
module type Stack = sig
  type 'a stack
  val empty : 'a stack (* the empty stack *)
  val is_empty : 'a stack -> bool
  val push : 'a -> 'a stack -> 'a stack
  val peek : 'a stack -> 'a
  val pop : 'a stack -> 'a stack
end
```

Stack I

- Here's the abstract definition (signature) of stack

```
module type Stack = sig
  type 'a stack
  val empty : 'a stack (* the empty stack *)
  val is_empty : 'a stack -> bool
  val push : 'a -> 'a stack -> 'a stack
  val peek : 'a stack -> 'a
  val pop : 'a stack -> 'a stack
end
```

- There are many ways to implement a stack
  - using lists
  - using variants
  - etc.

Stack II

- Using lists:

```
#use "stacksig.ml"
module ListStack : Stack =
  struct
    type 'a stack = 'a list
    let empty = []
    let is_empty s = (s = [])
    let push x s = x :: s
    let peek = fun s -> match s with
      | [] -> raise (Failure "ListStack.peek error")
      | x :: _ -> x
    let pop = fun s -> match s with
      | [] -> raise (Failure "ListStack.pop error")
      | _ :: xs -> xs
  end
```

Stack III

- Using variants:

```
#use "stacksig.ml"
module VariantsStack : Stack =
  struct
    type 'a stack =
      | Empty
      | Entry of 'a * 'a stack
    let empty = Empty
    let is_empty s = (s = Empty)
    let push x s = Entry (x, s)
    let peek = fun s -> match s with
      | Empty -> raise (Failure "VariantsStack.peek error")
      | Entry (x, _) -> x
    let pop = function
      | Empty -> raise (Failure "VariantsStack.pop error")
      | Entry (_, s) -> s
  end
```