## Lecture 11

# Modules

Iztok Savnik, FAMNIT

May, 2023.

# Literature

- Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, O’REILLY & Associates, 2000 (Chapter 14)
- John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapters 9)
- Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapter 9)

# Outline

1. Concept of module
2. Module as compilation unit
3. Modules in C
4. Modules in Java
5. Modules in Ocaml
6. Module language
7. Functors

# Concept of module

- Modular program design allows for decomposition of programs in more program units, called modules
- Module can be developed independently from the other parts of system
  - Modules can be compiled separately
  - Programmer does not need source code to work with some module
- Module interface defines the values, types, classes and functions that module offers to the user
  - Interface hides the implementation details
  - All that programmer needs to know is defined in interface

# Modules

- Modules are defined for some concrete “entity”
  - Examples of entities: a physical device, an menu object, a data structure, a functional unit of an application, ...
- Module defines a data environment
  - Set of data structures used for modelling the entity
  - Usually one data structure represents the module
  - Later defined as: ADT (Abstract Data Type)
- Module defines a set of operations
  - Operations that work with entity
  - Operations are usually defined on ADT instances

# Modules

- Module developer has considerable freedom in module implementation
  - Implementation may change completely while module interface stays the same
    - Module user can not notice the difference
  - From the other point of view, programmer does not need to know the details about other parts of the system to implement some module
    - It suffices they know the interface
  - Module interface hides implementation details that developer does not want to share

# Abstract Data Types

- In some PLs, modules are the same as ADTs
  - OCaml, ML, Modula II, ...
- What is an abstract data type (ADT)?
  - Special kind of data type!
    - ADT stands for a set of abstract data structures
    - We have a set of operations defined on a given data structure
  - Mathematical model
    - Algebra view: structures + a set of operations [+ set of rules]
  - ADT is “abstract” because it gives an implementation independent view
    - Internal representation of an ADT is hidden from the client
    - Operations manipulate abstract structures
  - Classes can also be seen as implementations of ADTs

# Modules

- Units of compilation
  - Isolation of code into single conceptual unit
  - Weak module language
- Module language
  - Module language that is part of programming language
  - Programming constructs and concepts for definition of module implementation and interface
- Functors
  - Parametrized modules
  - Modules can be parameters of modules
  - Generic code

# Module as compilation unit

- Many programming languages use modules as the compilation unit
  - C, C++, Java, Scala, Erlang, ML, Ocaml, Pascal, Modula, Perl, Python
- Module is usually represented by one or two files (can also be more than two)
  - C, C++: header and implementation files (mod.h, mod.c)
  - Java, Scala packages: classes (files) in directory
  - ML, Ocaml: two files (ocaml) or module language
  - Erlang: one or more files
  - Perl, Python: separate files

# Modules in C

- Program code is split into separate files
  - Files with *.c extension include parts of program
  - Files with *.h extension define module interfaces
- Benefits of using C modules
  - Program is divided into logically meaningful components
  - Separate components can be compiled into separate object files (later linked into one program)
  - Module is computational unit defined around some well defined concept
    - Data structure (e.g., stack), physical devices (e.g., driver), ...

# C module:

set.h
set.c

```
#ifndef SET_H
#define SET_H
/* Constants */
#define SET_LEN  sizeof(set_type) /* Maximal length of a set */
#define SET_MASK 0xffffffff   /* Mask for computing set op. */
#define MaxSetEl 30           /* max number of set elements. */
/* SET type definition */
typedef struct set_type {
  unsigned long lo;
  unsigned long hi;
} set_type;
/*----------------------- Exported functions -------------------------------*/
extern set_type* set_emp( set_type *S );
extern set_type* set_cpy( set_type *S, set_type *S1 );
extern set_type* set_union( set_type *S, set_type *S1, set_type *S2 );
extern set_type* set_intsc( set_type *S, set_type *S1, set_type *S2 );
extern set_type* set_diff( set_type *S, set_type *S1, set_type *S2 );
extern set_type* set_add( set_type *S, int el );
extern set_type* set_del( set_type *S, int el );
extern boolean   set_elm( int el, set_type *S );
extern boolean   set_subs( set_type *S1, set_type *S2 );
extern boolean   set_equ( set_type *S1, set_type *S2 );
extern int       set_card( set_type *S );
extern boolean   set_next_el( set_type *S, int cEl, int *nEl );
extern void      set_print( set_type *S );
#endif  /* SET_H */
```

```
#include <stdio.h>
#include "config.h"
#include "set.h"

/* Make set S empty */
set_type* set_emp( set_type *S )
{
  (*S).lo = 0; (*S).hi = 0;
  return S;
}/*set_emp*/

/* S = S1; */
set_type* set_cpy( set_type *S, set_type *S1 )
{
  (*S).lo = (*S1).lo;
  (*S).hi = (*S1).hi;
  return S;
}/*set_null*/

/* S = S1 + S2; */
set_type* set_union( set_type *S, set_type *S1, set_type *S2 )
{
  (*S).lo = (*S1).lo | (*S2).lo;
  (*S).hi = (*S1).hi | (*S2).hi;
  return S;
}
```

```
...

/* S = S – {el}; */
set_type* set_del( set_type *S, int el )
{
  if (el < 32) (*S).lo = (*S).lo & ((1 << el) ^ SET_MASK);
  Else (*S).hi = (*S).hi & ((1 << (el-32)) ^ SET_MASK);
  return S;
}

/* Membership test. */
boolean set_elm( int el, set_type *S )
{
  if (el < 32) return (( (*S).lo & (1 << el)) > 0);
  else return (( (*S).hi & (1 << (el-32))) > 0);
}

/* Subsumption test. */
boolean set_subs( set_type *S1, set_type *S2 )
{
  boolean Lo,Hi;
  Lo = (( (*S1).lo & (*S2).lo ) == (*S1).lo );
  Hi = (( (*S1).hi & (*S2).hi ) == (*S1).hi );
  return (Lo && Hi);
}
```

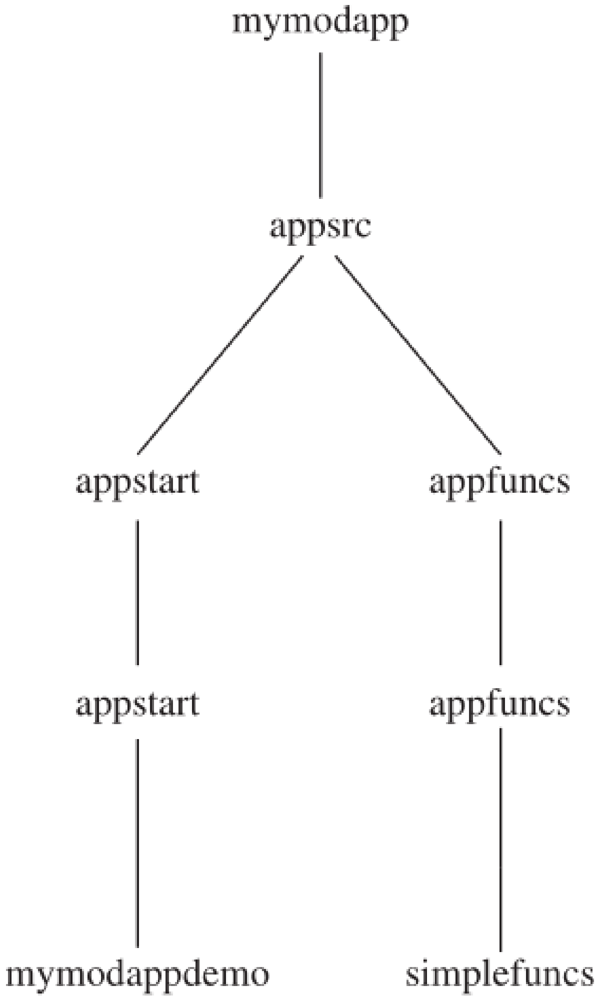
# Modules in Java

- Programs are organized as sets of packages
  - Members of a package are classes and interfaces
  - May include subpackages (recursively)
- Each package has its own set of names for classes and interfaces
  - Naming structure for packages is hierarchical
- If a set of packages is sufficiently cohesive, then packages may be grouped into a module.
  - Module can export some or all of its packages
  - Modul may depend (explicitely) on some other module
    - Then it can use packages from some other module

# Modules in Java

- Module declaration
  - A way to describe the relationships and dependencies of code that comprises application
  - module-info.java in module directory
- Module controls how its packages use other modules
  - By specifying dependences using `requires`
- Modules controls how other modules use its packages
  - Specifying which of its packages are exported using `exports`

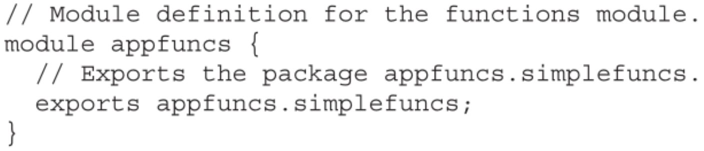
# Example: Java modules



## SimpleMathFuncs.java

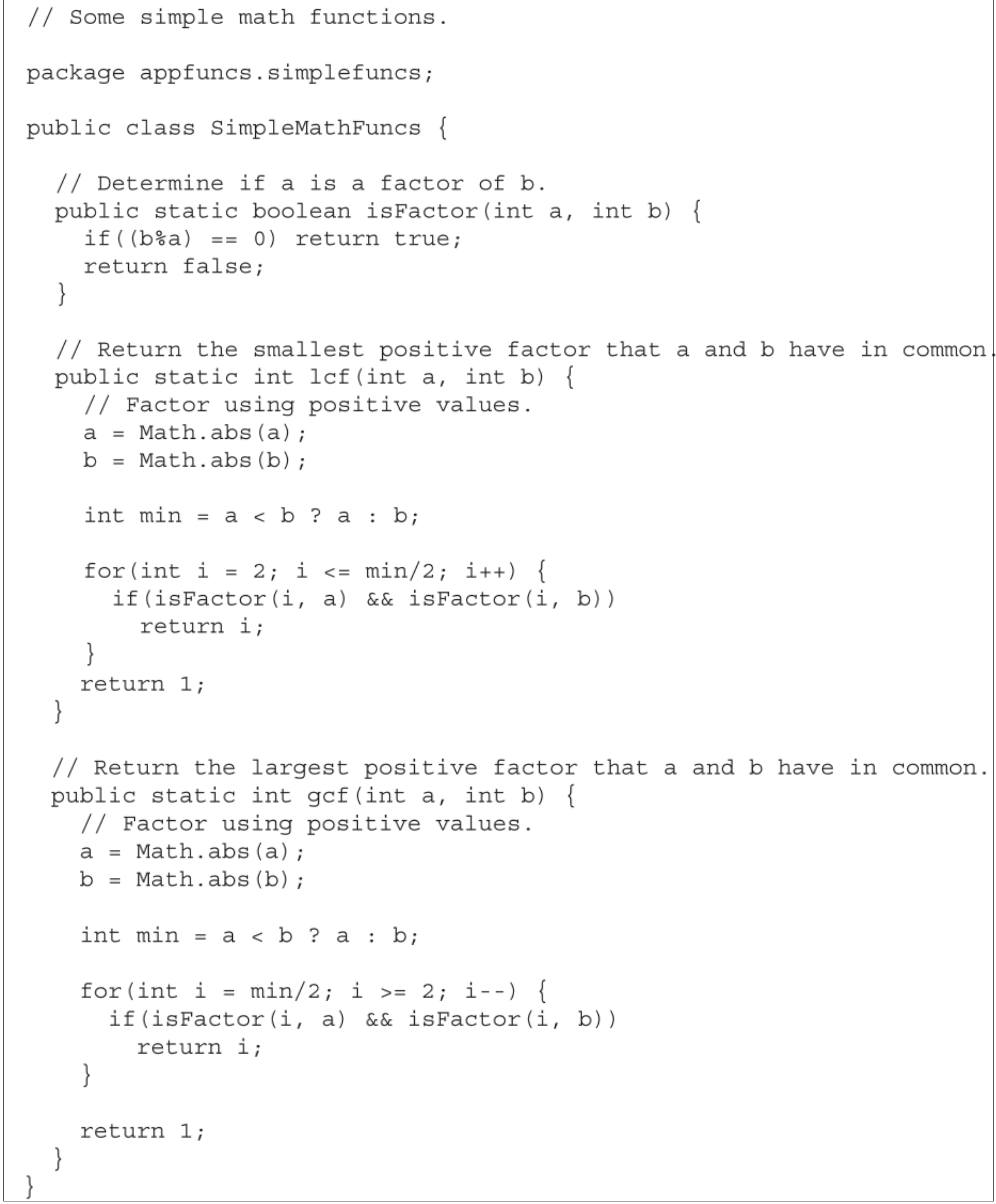
appsrc\appfuncs\appfuncs\simplefuncs

## module-info.java



```
// Module definition for the functions module.
module appfuncs {
  // Exports the package appfuncs.simplefuncs.
  exports appfuncs.simplefuncs;
}
```

appsrc\appfuncs

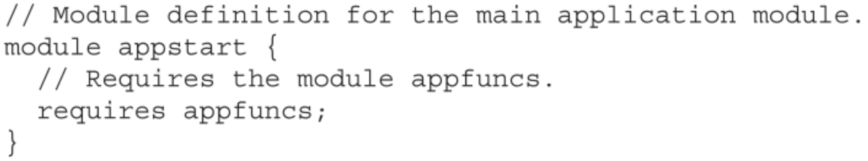


```
// Some simple math functions.

package appfuncs.simplefuncs;

public class SimpleMathFuncs {

  // Determine if a is a factor of b.
  public static boolean isFactor(int a, int b) {
    if((b%a) == 0) return true;
    return false;
  }

  // Return the smallest positive factor that a and b have in common.
  public static int lcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = 2; i <= min/2; i++) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }
    return 1;
  }

  // Return the largest positive factor that a and b have in common.
  public static int gcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = min/2; i >= 2; i--) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }

    return 1;
  }
}
```

# MyModAppDemo.java

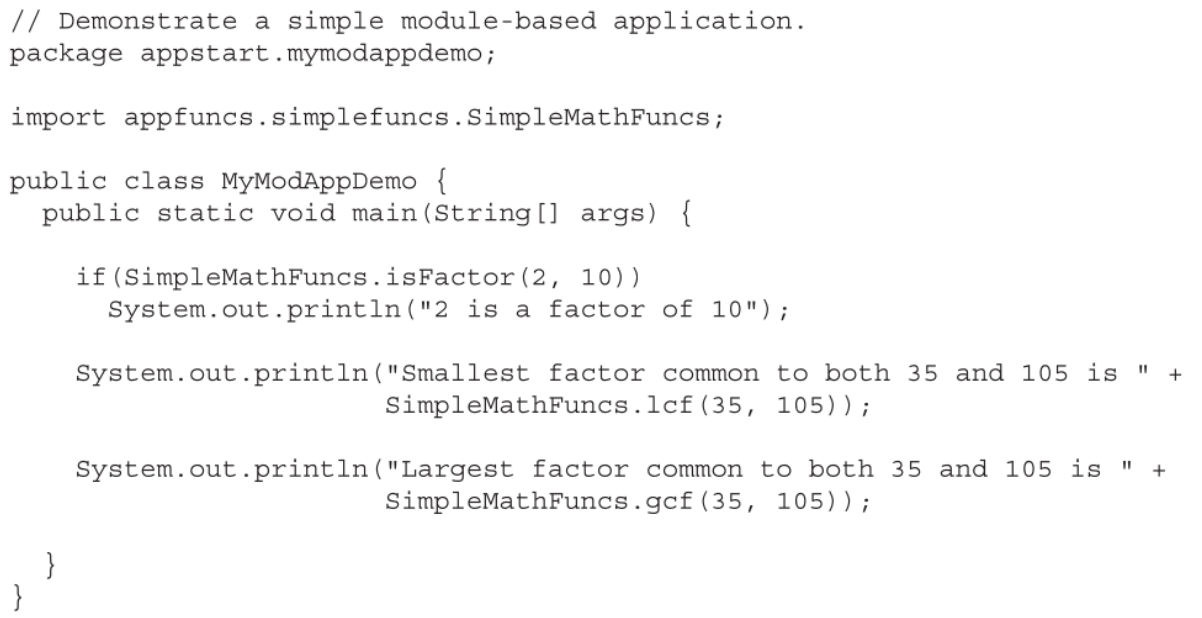
```
appsrc\appstart\appstart\mymodappdemo
```

# module-info.java



```
// Module definition for the main application module.
module appstart {
  // Requires the module appfuncs.
  requires appfuncs;
}
```

```
appsrc\appstart
```



```
// Demonstrate a simple module-based application.
package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;

public class MyModAppDemo {
  public static void main(String[] args) {

    if(SimpleMathFuncs.isFactor(2, 10))
      System.out.println("2 is a factor of 10");

    System.out.println("Smallest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.lcf(35, 105));

    System.out.println("Largest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.gcf(35, 105));

  }
}
```

# Modules in Ocaml

- Basic form of modules in Ocaml is very similar to modules in programming language C
  - File with code has extension “.ml”
  - Interface is a file with extension “.mli”
  - Similarly to C, one Ocaml file can also represent module
- First example
  - Module is stored in file Stack1.ml
  - Stack implementation is based on lists
  - This is also part of standary library

```
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with hd :: tl -> s.c <- tl; hd
                    |   []   -> raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c
```

# Modules in Ocaml

- Stack in the previous example is defined as abstract data type (abbr. ADT)
  - Module defines abstract data structure Stack1.t which is the center of module
  - First we create instance of Stack1.t and then we pass it as parameter to every function in module interface
  - ADTs share many similarities with classes in OO world
- Example of Stack module usage
  - Stack is created first
  - Using module functions
  - Code is stored in file Example1.ml

```
let s = Stack1.create ();;
Stack1.push 1 s; Stack1.push 2 s;
Stack1.push 3 s;;
let a = Stack1.pop s
and b = Stack1.pop s
and c = Stack1.pop s
in Printf.printf "Stack elements: %i, %i, %i\n" a b c;;
```

# Modules in Ocaml

- Accessing components of module
  - Components can be accessed using dot notation
    - Module.identifier
  - Module components can be imported into some other environment
    - open Module;;

```
open Stack1;;
let s = create ();;
push 1 s; push 2 s; push 3 s;;
let a = pop s and b = pop s
and c = pop s
in Printf.printf "Stack elements: \
                        %i, %i, %i\n" a b c;;
```

- Compilation
  - Program can be compiled using ocamlc compiler

```
$ ocamlc -o example1 Stack1.ml Example1.ml
$ ./example1
elements: 3, 2 and 1
```

# Module interface

- In the case module does not have associated interface file then <u>all components are public</u>
- Interface is used to <u>restrict</u> the access to module
  - Public components are listed in interface
  - Values and functions are defined as →

    ```
    val nom : type
    ```
  - Types defined in interface are <u>abstract</u>
- Example of module interface
  - File “Stack1.mli”
  - Users of Stack1 module do not have access to type Stack1.t
  - Access to instances of Stack1.t is possible through functions

```
type 'a t
exception Empty

val create: unit -> 'a t
val push: 'a -> 'a t -> unit
val pop: 'a t -> 'a
val clear : 'a t -> unit
val length: 'a t -> int
```

# Linking modules and interfaces

- Module is composed of two parts
  - Implementation that includes definitions of types, variables and functions
  - Interface that includes declarations of definitions that are be visible from outside
    - Interface can declare a subset of definitions !
    - Module can include auxiliary types, values and functions
    - These functions are not accessible from outside of module
- Declarations must be consistent with definitions
  - Interface can restrict types of components
- Interface is separated from implementation
  - We can have more implementations as well as more than one interfaces

# Linking modules and interfaces

- Module is stored in Stack2.ml
- Implementation of stack with array
- Interface can stay the same
- Module user does not need to know that the implementation changed

```
type 'a t = { mutable sp : int; mutable c : 'a array }
exception Empty
let create () = { sp=0 ; c = [||] }
let clear s = s.sp <- 0; s.c <- [||]
let size = 5
let increase s v =
  s.c <- Array.append s.c (Array.make size v)
let push x s =
  if s.sp >= Array.length s.c then increase s x;
  s.c.(s.sp) <- x;
  s.sp <- s.sp+1
let pop s =
  if s.sp = 0 then raise Empty
  else let () = s.sp <- s.sp-1 in s.c.(s.sp)
let length s = s.sp
let iter f s = for i = s.sp-1 downto 0 do f s.c.(i) done
```

# Linking modules and interfaces

- Example of linking one interface and two implementations
  - Stack1.mli and Stack2.mli are the same
    - In module language will will be able to reuse Stack1.mli
  - example{1|2}.ml differ solely in open Stack{1|2}

```
$ ocamlc -o example1 Stack1.mli Stack1.ml example1.ml
$ ./example1
Stack elements: 3, 2, 1
$ ocamlc -o example2 Stack2.mli Stack2.ml example2.ml
$ ./example2
Stack elements: 3, 2, 1
```

# Module language in Ocaml

- Module language is a subset of programming language constructs that deals with modules
- Module interface is called signature and module implementation is called structure
- Syntax for definition of module signature and structure
  - Module name must start with uppercase letter

```
module type NAME = sig declarations end
module Name = struct definitions end
```

- Module signature and structure do not need to be named
  - Anonymous signature and structure

```
sig declarations end
struct definitions end
```

# Module language

- Module definition in Ocaml

```
module Name : signature = structure
module Name = (structure : signature)
```

- Any structure has default signature
  - Declarations of all definitions from structure
  - The most general type is used in declarations
- Type checking procedure verifies signature and structure
  - Definitions in structure can be more general
  - Signature can restrict default signature of structure

# Example of module

- Data structure includes pair of lists
- Queue + Stack

```
# module PairOfLists = struct
  type 'a t = ('a list * 'a list) ref
  exception Empty
  let create () = ref ([], [])

  let enqueue x queue =
   let front, back = !queue in
     queue := (x::front, back)

  let rec dequeue queue =
   match !queue with
    (front, x :: back) -> queue := (front, back);  x
   | ([], []) -> raise Empty
   | (front, []) -> queue := ([], List.rev front);
              dequeue queue

  let push x queue = enqueue x queue

  let rec pop queue =
   match !queue with
    (x::front,back) -> queue := (front,back); x
   | ([],[]) -> raise Empty
   | ([],back) -> queue := (List.rev back,[]);
           pop queue
 end;;
```

```
# let q = PairOfLists.create ();;
val q : ('_a list * '_b list) ref = {contents = ([], [])}
# PairOfLists.enqueue 1 q; PairOfLists.push 2 q;;
- : unit = ()
# q;;
- : (int list * '_a list) ref = {contents = ([2; 1], [])}
# PairOfLists.dequeue q;;
- : int = 1
# q;;
- : (int list * int list) ref = {contents = ([], [2])}
# PairOfLists.pop q;;
- : int = 2
```

# Information hiding

- Hiding abstract type of module
- Example of Stack
  - When stack is created Ocaml annotates it as abstract
  - Now, Stack1 user can not see the definition of Stack.t
  - Stack.t is hidden by signature
- Stack instance can be manipulated solely through module functions

```
# module type Stack =
    sig
      type 'a t
      exception Empty
      val create: unit -> 'a t
      val push: 'a -> 'a t -> unit
      val pop: 'a t -> 'a
    end ;;
# module Stack1 = (PairOfLists:Stack);;
module Stack1 : Stack
```

```
# let s = Stack1.create ();;
val s : '_a Stack1.t = <abstr>
# Stack1.push 1 s;;
- : unit = ()
# Stack1.push 2 s;;
- : unit = ()
# Stack1.pop s;;
- : int = 2
# Stack1.pop s;;
- : int = 1
```

# Multiple views of module

- Restricting modules with signatures allows creation of different views to a single structure
  - For instance, we can define another view of PairOfLists
  - The same module is used for implementation of stack and queue

```
# module type Queue =
    sig
      type 'a t
      exception Empty
      val create: unit -> 'a t
      val enqueue: 'a -> 'a t -> unit
      val dequeue: 'a t -> 'a
    end ;;
```

```
# module Queue1 = (PairOfLists:Queue);;
module Queue1 : Queue
# let v = Queue1.create ();;
val v : '_a Queue1.t = <abstr>
# Queue1.enqueue 1 v; Queue1.enqueue 2 v;;
- : unit = ()
# Queue1.dequeue v;;
- : int = 1
# Queue1.dequeue v;;
- : int = 2
```

# Priority queue

- Smallest element is at the top
- Function insert rotates left and right sub-trees

```
# module PrioQueue =
  struct
    type priority = int
    type 'a queue = Empty | Node of priority * 'a * 'a queue * 'a queue
    let empty = Empty
    let rec insert queue prio elt =
      match queue with
        Empty -> Node(prio, elt, Empty, Empty)
      | Node(p, e, left, right) ->
          if prio <= p
          then Node(prio, elt, insert right p e, left)
          else Node(p, e, insert right prio elt, left)
    exception Queue_is_empty
    let rec remove_top = function
        Empty -> raise Queue_is_empty
      | Node(prio, elt, left, Empty) -> left
      | Node(prio, elt, Empty, right) -> right
      | Node(prio, elt, (Node(lprio, lelt, _, _) as left),
                         (Node(rprio, relt, _, _) as right)) ->
          if lprio <= rprio
          then Node(lprio, lelt, remove_top left, right)
          else Node(rprio, relt, left, remove_top right)
    let extract = function
        Empty -> raise Queue_is_empty
      | Node(prio, elt, _, _) as queue -> (prio, elt, remove_top queue)
  end;;
```

```
# PrioQueue.insert PrioQueue.empty 1 "hello";;
- : string PrioQueue.queue =
PrioQueue.Node (1, "hello", PrioQueue.Empty,
PrioQueue.Empty)
```

# Parametrized modules

- Parametrized modules or functors
- Parametrized modules are generic modules based on modules that are passed as the arguments
  - ADT can be passed as parameter to host module to provide the data type and operations defined by parameter ADT to host
  - Relation between parametized modules and modules are similar to relationship between functions and values
  - As function call constructs new value, invocation of parametrized module constructs new module
  - Functors are functions from modules to modules

# Parametrized modules

- Functors extend programming language with constructs that improve the reusability of code
- Syntax of functor definition is close to syntax of function definition
  - Including abbr.

```
functor ( Name : signature ) -> structure
module Name1 ( Name2 : signature ) = structure
```

- Example of simple functor Couple that uses parameter module Q
  - Q defines type Q.t
  - Couple defines type Couple.t = Q.t * Q.t

```
# module Couple = functor ( Q : sig type t end ) ->
    struct type couple = Q.t * Q.t end ;;
```

# Parametrized modules

- Functor with more than one parameter modules
  - Basic syntax
  - Abbreviated syntax

```
functor ( Name1 : signature1 ) ->
          ...
          functor ( Namen : signaturen ) -> structure
```

```
module Name (Name1:signature1) . . . (Namen:signaturen) = structure
```

  - Functor application

```
module Name = <functor> ( structure1 ) . . . ( structuren )
```

- Closed functor is a functor that does not reference other functors but parameters
  - The use of closed functors improve genericity of code

# Example of functor

```
# type comparison = Less | Equal |
                     Greater;;
type comparison = Less | Equal | Greater
# module type ORDERED_TYPE =
   sig
     type t
     val compare: t -> t -> comparison
   end;;
```

```
# module Set =
   functor (Elt: ORDERED_TYPE) ->
     struct
       type element = Elt.t
       type set = element list
       let empty = []
       let rec add x s =
         match s with
           [] -> [x]
         | hd::tl ->
            match Elt.compare x hd with
              Equal   -> s        (* x is already in s *)
            | Less    -> x :: s    (* x is smaller than all elements of s *)
            | Greater -> hd :: add x tl
       let rec member x s =
         match s with
           [] -> false
         | hd::tl ->
             match Elt.compare x hd with
               Equal   -> true     (* x belongs to s *)
             | Less    -> false    (* x is smaller than all elements of s *)
             | Greater -> member x tl
     end;;
```

```
# module OrderedString =
   struct
     type t = string
     let compare x y = if x=y then Equal else if x < y then Less else Greater
   end;;
module OrderedString :
  sig type t = string val compare : 'a -> 'a -> comparison end

# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString.t list
    val member : OrderedString.t -> OrderedString.t list -> bool
  end

# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
- : bool = false
```