Chapter 8

Some Examples: M3-- in Action

As an example of the possibilities of `m3--`, we will first look at some simple applications of generic types. The second section compares implementations of lists in different languages. The last section show an implementation of generic queues (or sequences) in `m3--`.

8.1 Some Examples of Code in `m3--`

For reusable code which can be made generic, it is important that the polymorphism of the algorithms can be made explicit and that the manner and degree of this polymorphism is visible in both specification and implementation. In Modula-3[39] polymorphic procedures can be written, but the polymorphism is not as explicit as one would like. Explicit polymorphism, as is possible in `m3--`, will also allow the compiler to generate more optimal (leaner) code in those situations where polymorphism is not needed.

One of the basic tenets is: if it is possible to make polymorphism explicit in the program text, code should only be polymorphic if specified as such. This means (e.g.) that a pointer to some object type cannot be passed as if it were a pointer to some ancestor type, unless explicitly allowed.

Having to write out all those polymorphic type expressions may seem bothersome, but the presence of generic (read: parameterized) types allows some shorthand notations to be defined, as shown in listing 8.1.

Listing 8.1: The `GENERICS` Interface

```
 INTERFACE GENERICS;

TYPE
  (*Open Arrays: *)
  OpenArray = FORALL Elem  <: ANY IN
              EXISTS Index <: ORDINAL IN
              ARRAY Index OF Elem;

  (*Polymorphic Versions of Types: *)
  SomeSubtypeOf = FORALL T <: ANY IN
                  EXISTS Sub <: T IN Sub;

  UnionSubtypeOf  = FORALL T <: ANY IN
```

```
                      UNION  Sub <: T IN Sub;

    (*A Polymorphic Type to a Type: *)
    SomeRefOf = FORALL T <: ANY IN
                EXISTS Sub <: T IN REF T;

    UnionRefOf = FORALL T <: ANY IN
                 UNION Sub <: T IN REF T;

    (*Polymorphic Versions of Generic Types *)
    Opaque = FORALL Generic <: ANY IN
             EXISTS Sub <: BOUND(Generic) IN Generic(Sub);

    Union  = FORALL Generic <: ANY IN
             UNION  Sub <: BOUND(Generic) IN Generic(Sub);

  END GENERICS.
```

(End of Listing 8.1)

Using the types from this interface a standard group of types can be declared for each new object type, for example:

```
  TYPE
      T = OBJECT ... END;
     RT = REF T;
    URT = GENERICS.UnionRefOf(T);
     ET = GENERICS.SomeSubtypeOf(T);
```

— `RT` is now the type of a pointer to `T`,
— `URT` is the type of a polymorphic variable for a pointer to some subtype of `T`, and
— `ET` is the type for an (implicit) polymorphic parameter of some subtype of `T`.

A second example is the `Array` interface shown in listing 8.2, which provides several generic procedures for array allocation and copying.

Listing 8.2: The `Array` Interface

```
   INTERFACE Array;

  CONST
    New: FORALL Index <: ORDINAL IN FORALL Elem  <: ANY IN
         PROCEDURE (CONST low, high: Index
                  ): EXISTS low2High <: Index IN
                     REF ARRAY low2High OF Elem;
(*   The call New(IndexBase)(ElementType)(low, high) returns a pointer to
  an array. The array's element type is Elem, and the index type is
  the subrange type [low..high], where low and high have type
  IndexBase.
*)
```

```
    CopyUp: FORALL Index <: ORDINAL IN FORALL Elem <: ANY IN
            PROCEDURE(CONST source: EXISTS actual <: Index IN
                                    ARRAY actual OF Elem;
                      CONST srcLow, srcHigh: Index;
                      VAR   dest:   EXISTS actual <: Index IN
                                    ARRAY actual OF Elem;
                      CONST dstLow: Index);
(*  The procedure CopyUp copies one array to another in lowerbound to
  upperbound order. Its parameters are:
    Index    The base type of the index types of both arrays.
    Elem     The element type of both arrays.
    source   The array to be copied.
    srcLow,
    srcHigh  The bounds of the subarray to be copied.
    dest     The destination array.
    dstLow   The location where the "src [srcLow]" should be copied to.
*)

    CopyDown: FORALL Index <: ORDINAL IN FORALL Elem <: ANY IN
              PROCEDURE(CONST source: EXISTS actual <: Index IN
                                      ARRAY actual OF Elem;
                        CONST srcLow, srcHigh: Index;
                        VAR   dest:   EXISTS actual <: Index IN
                                      ARRAY actual OF Elem;
                        CONST dstLow: Index);
(*  CopyDown also copies array, but now performs the copying in the
  reverse order. Having both copy functions allows in-array copies when
  source and destination would overlap.
*)
  END Array.
```

(End of Listing 8.2)

8.2 Two Approaches to Linked Lists

In this section I will present two different approaches to linked lists, respectively from the standard Modula-3 library[26] and the standard Modular Pascal library[13]. Both will be followed by a comparable implementation in `m3--`, be it with a stricter typing.

8.2.1 The Generic Approach to Lists

In the Modula-3 standard library the generic `List` interface and module provide for linked lists and basic operations on them. Following Modula-3 tradition, the parameter interface for the list element exports a type called `T` and a comparison function called `Equal` for comparing elements.

Listing 8.3: The Modula-3 `List` Interface

```
GENERIC INTERFACE List(Elem);
(* Where "Elem.T" is not an open array type and "Elem" contains
 *
 *   PROCEDURE Equal(k1, k2: Elem.T): BOOLEAN;
 *
 *  "Equal" may be declared with a parameter mode of either
 *  "VALUE" or "READONLY", but not "VAR".
 *)

TYPE T = OBJECT head: Elem.T; tail: T END;

(* A "List.T" represents a linked list of items of type    *
 * "Elem.T".                                               *
 * None of the operations of this interface modify the     *
 * "head" field of an existing list element.               *)

PROCEDURE Cons(READONLY hd: Elem.T; tl: T): T;
(* Equivalent to "NEW(T, head := hd, tail := tl)".         *)

PROCEDURE List1(READONLY e1: Elem.T): T;
PROCEDURE List2(READONLY e1, e2: Elem.T): T;
PROCEDURE List3(READONLY e1, e2, e3: Elem.T): T;
(* Return lists containing one, two or three "Elem.T"s.    *)

PROCEDURE FromArray(READONLY ar: ARRAY OF Elem.T): T;
(* Return a list containing the elements of "e" in index.  *)

PROCEDURE Length(l: T): CARDINAL;
(* Return the number of elements of "l".                   *)

PROCEDURE Nth(l: T; n: CARDINAL): Elem.T;
(* Return element "n" of list "l".                         *
 * Element 0 is "l.head", element 1 is "l.tail.head", etc. *
 * Cause a checked runtime error if "n >= Length(l)".      *)

PROCEDURE Member(l: T; READONLY e: Elem.T): BOOLEAN;
(* Return "TRUE" if some element of "l" is equal to "e",   *
 *       else return "FALSE".                              *
 * The comparison is performed by "Elem.Equal".            *)

PROCEDURE Append(l1: T; l2: T): T;
(* Append two lists together, returning the new list.      *
 * "Append" does this by copying of the cells of "l1".     *)

PROCEDURE Reverse(l: T): T;
(* Return a list containing the elements of "l" in reverse *
 * order. "Reverse" copies the cells.                      *)

END List.
```

(End of Listing 8.3)

Next follows a similar `List` interface in `m3--`, using a generic record type for `List.T`. All functions for lists are generic in the element type also, although in a few cases the element type can be made an implicit type parameter using existential types.

Listing 8.4: A Generic `List` Interface in `m3--`

```
INTERFACE List;

IMPORT GENERICS

(* The "List" interface provides operations on linked lists *
 * of arbitrary element types. "List.T" is generic in the   *
 * element type.                                            *)

TYPE
  T = FORALL Elem <: ANY IN
      RECORD
        head: Elem;
(* "head" is the actual element value of the list object.   *)

        tail: REF T(Elem)
(* By declaring "tail" using the instantiated version of    *
 * "T", the list is garanteed to be homogeneous.            *)
      END;

CONST
(* "Cons" returns a new "T".                                *)
  Cons : FORALL Elem <: ANY IN
         PROCEDURE (CONST hd : Elem;
                    CONST tl : T(Elem)
                   ): T(Elem);

(* "List1", "List2" and "List3" are ‘shorthands’ for        *
 * creating lists of one, two or three elements.            *)
  List1 : FORALL Elem <: ANY IN
          PROCEDURE (CONST e1 : Elem): T(Elem);
  List2 : FORALL Elem <: ANY IN
          PROCEDURE (CONST e1, e2 : Elem): T(Elem);
  List3 : FORALL Elem <: ANY IN
          PROCEDURE (CONST e1, e2, e3 : Elem): T(Elem);

(* "FromArray" creates a new "T(Elem)" from an array.       *)
```

```
  FromArray
    : FORALL Elem <: ANY IN
      PROCEDURE (CONST ar : GENERICS.OpenArray(Elem)
                ): T(Elem);

(* "Length" returns the number of elements in a given list. *
 * It does not need an explicit instantiation.             *)
  Length : PROCEDURE (CONST l : EXISTS Elem <: ANY IN
                              T(Elem)): CARDINAL;

(* "Nth" returns the "n"th element in the given list.       *)
  Nth : FORALL Elem <: ANY IN
        PROCEDURE (CONST l: T(Elem);
                   CONST n: CARDINAL
                  ): Elem;

(* "Member" returns "TRUE" if the given element value is    *
 * present in the list, using the given comparison function.*)
  Member
    : FORALL Elem <: ANY IN
      PROCEDURE (CONST l: T(Elem); CONST e: Elem;
                 CONST equal: PROCEDURE (CONST a, b: Elem
                                        ): BOOLEAN
                ): BOOLEAN;

(* "Append" appends "l2" after a copy of "l1".              *)
  Append : FORALL Elem <: ANY IN
           PROCEDURE (CONST l1, l2: T(Elem)): T(Elem);

(* "Reverse" returns a copy of "l" in reverse order.        *)
  Reverse : FORALL Elem <: ANY IN
            PROCEDURE(CONST l: T(Elem)): T(Elem);

END List .
```

(End of Listing 8.4)

8.2.2 Lists Using Inheritance

In Modular Pascal generic types or modules are not possible, but a limited form of type extension can be used. The assignability rules in ModPas are such that a value of a pointer to some record type is assignable to a pointer type which has the record's first element as its element type. For example, in the `lists` module shown below the type `list` is declared as a pointer to a pointer of that same type:

```
TYPE
  list = ^list;
```

A user of the `lists` module might declare:

```
TYPE
  intList = ^intRec;
  intRec  = RECORD
              next: list;
              val : INTEGER
            END;
```

The pointer type `intList` is declared here as a pointer to a record whose first field is a `list`. Thanks to the modified assignability rules any pointer of type `intList` may now be assigned to a `list` variable, or passed as a `list` parameter. For the reverse assignment the pointer must be cast, using the `intList` typename as if it were a function. This is a form of inheritance as found with object oriented programming languages.

I will now discuss the actual `lists` module. In the listing below only the declarations and procedure headings are shown. For the sake of brevity the bodies are omitted. The following conventions are adopted:

- In general, no distinction is made between the types of a list element and the list with that element as its head. If necessary, the distinction is denoted by a suitable choice of names of the formal parameters.
- When a list operation potentially changes a data-structure that is passed from an application, the result will be preferably be delivered as a function result (of the generic list type).

Listing 8.5: The Modular Pascal `lists` Module

```
MODULE lists;

TYPE list = ^list;

FUNCTION to_list(l: list; elt: list): list;
(* POST: to_list := elt & l                                *)

FUNCTION from_list(VAR l: list): list;
(* PRE: l <> NIL                                           *)

TYPE fifo_list = RECORD head, tail: list END;

PROCEDURE init_fifo(VAR f: fifo_list);
(* POST: is_empty(f)                                       *)

PROCEDURE to_fifo(VAR f: fifo_list; elt: list);
(* PRE: f = f0, POST: f = f0 & elt                         *)
PROCEDURE from_fifo(VAR f: fifo_list): list;

FUNCTION search_list(l: list;
                       FUNCTION crit(el: list): boolean
```

```
                      ): list;
  (* POST: let s0 = search_list(l, crit), then                *
   *       (s0 <> NIL) => (s0 IN l) AND crit(s0)              *)

  PROCEDURE for_all_in_list(l: list;
                            PROCEDURE action(el: list)
                           );

  FUNCTION filter_list(l: list;
                       FUNCTION is_to_be_removed(el: list
                                                ): boolean;
                       PROCEDURE last_action(el: list)
                      ): list;
  (* POST: All elements in l that satisfy "is_to_be_removed"  *
   *       have been removed from the result list, and for    *
   *       each of them "last_action" has been called.        *)

  FUNCTION insert_in_list(l: list; el: list;
                          FUNCTION before(el1, el2: list): boolean
                         ): list;

  FUNCTION merge_lists(l1, l2: list;
                       FUNCTION before(el1, el2: list): boolean
                      ): list;
  (* PRE: (l1 <> NIL) AND (l2 <> NIL)                         *)

  FUNCTION sort_list(t :list;
                     FUNCTION before(l1, l2: list): boolean
                    ): list;


  BEGIN END (* of MODULE lists *) .
```

(End of Listing 8.5)

The first data type exported is the bare `list`. The first function (`to_list()`) corresponds to the `Cons()` function of the previous two interfaces, with the difference that the list concept is 'packaged' into the datastructure. I.e. there is no separate 'element' type. The user must construct his data structures so as to match the list structure discussed above. The second function (`from_list()`) receives the list as a var parameter and removes its first element. This first element is then returned.

The second data structure exported is `fifo_list`, together with a procedure to initialize it (`init_fifo()`). Two functions are also given for adding an element to (`to_fifo()`,) and removing an element (`from_fifo()`) from the list.

The next three functions are: `search_list()`, which returns the first element of the given list for which a given function returns `TRUE`, or `NIL`. `for_all_in_list()` acts like a `for` statement, iterating over all elements of a list, and `filter_list()`, which will selectively remove elements from a list and return the remainder.

The last three functions concern ordered lists. For each of them the ordering is described in terms of a boolean function `before()` which the programmer must pass to the library function. It is the responsibility of the programmer to make sure the same ordering function is used with each call. The three functions concerned with ordered lists are: `insert_in_list()`, which inserts an element in the list while preserving its order, `merge_list()`, which merges two ordered lists, and `sort_list()`, which sorts a list.

A big difference with the generic approach used in the Modula-3 implementation is the 'cleanness' of this module. The actual element type has effectively been kept out of the 'picture'. The cost of it all is of course some added complexity in the usage of this module, where the basic `list` type must be cast back to the specific version. E.g. this must be done in the user-supplied `before()` function. Also the user is given the burden of ensuring the applicability of parameter functions to the specific list types used. This was discussed in section 4.1.2.

The following interface shows an interface for a module providing a *comparable* functionality, in `m3--`. The list defined below is guaranteed to be homogeneous. The interface (`List2`) exports not only *the* list type `List2.T` as an opaque object type, but also a non-opaque object type `List2.Public`, which is the visible ancestor of `List2.T`. The user of interface `List2` only has the information found in the declaration of `List2.Public`, the complete implementation of `List2.T` is hidden.

Listing 8.6: Lists Using Inheritance in `m3--`

```
INTERFACE List2;

IMPORT GENERICS;

TYPE
  T <: Public;
(* A "List2.T" implements a linked list. It must be extended  *
 * with an actual ‘element’ value.                              *)

  Public = OBJECT
    next: REF Current   (* "List2.T"s are homogeneous lists.  *)
  METHODS
    to_list(CONST elt: REF Current): REF Current;
(* Prepends ‘‘elt’’ to the list and returns the resul. *)

    search(CONST crit: PROCEDURE (CONST x: Current): BOOLEAN
           ): REF Current;
(* Returns the element "x" for which "crit(x)" returns "TRUE" *)

    forall(CONST action: PROCEDURE (VAR x: Current))
(* Calls "action()" for each element of the list              *)

  END;

CONST
```

```
    from_list: FORALL list <: T IN
               PROCEDURE (VAR l: REF list): REF list;
  (* Removes the first element of "l" and returns this element  *)

    FIFO <: PublicFIFO;
  (* A "List2.FIFO" is a new type, implementing a FIFO list.    *)

    PublicFIFO = OBJECT
      head,
      tail: GENERICS.UnionRefOf(T)
    METHODS
      init();
  (* "init()" initializes the FIFO.                             *)

      append(CONST elt: GENERICS.SomeRefOf(T))
  (* "append()" adds an element to the end of the list.         *)

    END;

    Sorted <: SortedPublic;
  (* A "List2.Sorted" implements a sorted list. Extensions must *
   * override the "before" method to determine the ordering.    *)

    SortedPublic = T OBJECT
    METHODS
      before(CONST x: Current): BOOLEAN;
  (* "before" returns "TRUE" if "Self" should come before "x".  *)

      insert(CONST elt: REF Current);
  (* Inserts "elt" guided by the ordering function "before".    *)

      merge(CONST l: REF Current);
  (* Merges two lists, preserving order.                        *)

    END;

  PROCEDURE sort(CONST list: REF T): REF Sorted;
  (* Returns a "Sorted" copy of the list.                       *)

  END List2.
```

(End of Listing 8.6)

Here the polymorphic type `Current` is used to provide for the necessary enforcment of equal types, and to guarantee the types of the returned values.

8.3 Generic Queues

In this section I will give a larger example of an m3-- module and interface. They implement generic queue objects, approaching the generality problem with both parameterization and inheritance. The chosen form allows parameterization for the instantiation of a queue object with an actual element type. Inheritance can be used (without instantiating, if desired) to extend the queues with a queueing strategy.

Listing 8.7: The Queue Interface in m3--

```
INTERFACE Queue;
(* The "Queue" interface exports a generic queue object.     *)

TYPE
  T <: Public;

(* "Queue.Public" is a generic type, and can be instantiated *
 * with an actual element type. Note that the subtyping      *
 * rules allow us to know that "Queue.T" itself is also      *
 * generic with the same bound.                              *)

  Public = FORALL Elem <: ANY IN
           OBJECT
(* The queue will be implemented using an object type.       *
 * The queueing strategy is not fixed, and can be changed by *
 * overriding the available methods.                         *)

           METHODS
             init(CONST sizeHint: CARDINAL);
           (* Performs the "Queue.T" level initialization    *)

             isempty(): BOOLEAN;
           (* Returns "TRUE" is the queue is empty           *)

             length(): CARDINAL;
           (* Returns the number of elements in the queue    *)

             get(CONST i: CARDINAL): Elem;
           (* Return the "i"th element element in the queue  *)

             put(CONST i: CARDINAL; CONST e: Elem);
           (* Put "e" in the queue as the "i"th element      *)

             delete(CONST i: CARDINAL);
           (* Remove the "i"th element from the queue        *)

             insert(CONST i: CARDINAL; CONST e: Elem);
```

```
            (* Insert "e" to become the new "i"th element     *)

            END;

  END Queue.
```

(End of Listing 8.7)

8.3.1 The Queue module

All the method procedures declared in this module are generic in both the element type *and* the object type. The user of this module will never notice this, since the instantiation of the object type will cause the compiler to instantiate the method procedures, and method selection will cause instantiation with the object type.

Listing 8.8: The `Queue` Module in `m3--`

```
  MODULE Queue;

  IMPORT Array;

  (* This implementation uses an array on the heap to store    *
   * the actual queue elements. The following constants are    *
   * used for the initial size of the array and the number of  *
   * elements added when expansion is needed.                  *)
  CONST
    minSize  = 32;
    sizeStep = 32;


  (* Here "T" is revealed:                                     *)
  REVEAL
    T = FORALL Elem <: ANY IN
        Public(Elem) OBJECT
          ar : UNION Index <: CARDINAL IN
                    REF ARRAY Index OF Elem;
        (* The "ar" field points to the array containing the   *
         * actual queue elements.                              *)

          size : CARDINAL;
        (* The current size of "ar" is kept in "size"          *)

          length : CARDINAL;
        (* The number of elements actually used in "ar"        *
         * Invariant: "length" < "size".                       *)

        OVERRIDES
        (* The following lines link implementations to methods *)
        init    := Init;
        isempty := isEmpty;
        length  := Length;
        get     := Get;
        put     := Put;
        delete  := Delete;
        insert  := Insert;
      END;

  (* The method procedures are all generic in both the element *
   * type and the "Current" type. Instantiation will be        *
   * performed by the compiler.                                *)
  CONST

  (* "Init()" initializes several fields.                      *)
    Init= FORALL Elem <: ANY IN
          FORALL Current <: T(Elem) IN
          PROCEDURE (VAR   Self    : Current;
                     CONST sizeHint: CARDINAL) {
    BEGIN
      IF sizeHint < minSize THEN
        sizeHint := minSize;
      END;
      Self.ar     := Array.New(CARDINAL)(Elem)(0, sizeHint);
      Self.size   := sizeHint;
      Self.length := 0;
    END };

  (* "isEmpty()" returns "TRUE" if the "length" field is 0     *)
    isEmpty= FORALL Elem <: ANY IN
             FORALL Current <: T(Elem) IN
             PROCEDURE (VAR Self: Current): BOOLEAN {
    BEGIN
      RETURN Self.length = 0;
    END };

  (* "Length()" returns the value of the "length" field        *)
    Length= FORALL Elem <: ANY IN
            FORALL Current <: T(Elem) IN
            PROCEDURE (VAR Self: Current): CARDINAL {
    BEGIN
      RETURN Self.length;
    END };

  (* "Get()" returns the "i"th element                         *)
    Get= FORALL Elem <: ANY IN
```

```
       FORALL Current <: T(Elem) IN
       PROCEDURE (VAR   Self: Current;
                  CONST i   : CARDINAL): Elem {
  BEGIN
    RETURN Self.ar^ [i];
  END };

(* "Put()" stores a value in the array.                    *
 * "Array.CopyUp()" is used to copy the contents of the old  *
 * array if the array has to be expanded.                  *)
  Put= FORALL Elem <: ANY IN
       FORALL Current <: T(Elem) IN
       PROCEDURE (VAR   Self: Current;
                  CONST i   : CARDINAL;
                  CONST e   : Elem) {
  VAR
    n: CARDINAL;
    a: UNION Index <: CARDINAL IN REF ARRAY Index OF Elem;
  BEGIN
    IF i > Self.size THEN
      n := ((i DIV sizeStep)+1) * sizeStep;
      a := Array.New(CARDINAL)(Elem)(1, n);
      Array.CopyUp(CARDINAL)(Elem)
                  (Self.ar^, 1, Self.length, a^, 1);
      Self.ar := a;
      Self.size := n;
    END;
    Self.ar^ [i] := e;
    IF i > Self.length THEN
      Self.length := i;
    END;
  END };

(* "Delete()" removes an element using "Array.CopyUp()"     *)
  Delete= FORALL Elem <: ANY IN
          FORALL Current <: T(Elem) IN
          PROCEDURE (VAR   Self: Current;
                     CONST i   : CARDINAL) {
  BEGIN
    Array.CopyUp(CARDINAL)(Elem)
                (Self.ar^, i+1, Self.length, Self.ar^, i);
    Self.length := Self.length - 1;
  END };

(* "Insert()" adds an element.                              *
 * Again this may lead to the creation of a bigger array.   *)
```

```
    Insert= FORALL Elem <: ANY IN
            FORALL Current <: T(Elem) IN
            PROCEDURE (VAR   Self: Current;
                       CONST i   : CARDINAL;
                       CONST e   : Elem) {
    VAR
      n: CARDINAL;
      a: UNION Index <: CARDINAL IN REF ARRAY Index OF Elem;
    BEGIN
      IF Self.length = Self.size THEN
        n := Self.size + sizeStep;
        a := Array.New(CARDINAL)(Elem)(1, n);
        Array.CopyUp(CARDINAL)(Elem)
                    (Self.ar^, 1, Self.size, a^, 1);
        Self.ar := a;
        Self.size := n;
      END;
      IF i < Self.length THEN
        Array.CopyDown(CARDINAL)(Elem)
                      (Self.ar^, i, Self.length,
                       Self.ar^, i+1);
      END;
      Self.length  := Self.length + 1;
      Self.ar^ [i] := e;
    END };

  END Queue.
```

(End of Listing 8.8)

8.3.2 An Interface for FIFO Queues

Using the `Queue` interface and module FIFO queues can be implemented using inheritance. 'New' operations would then be push and pop, with the implementation providing the FIFO functionality. Below an example of how an interface for such a FIFO module could be constructed:

Listing 8.9: A `FIFO` Interface using `Queue`

```
  INTERFACE FIFO;

  IMPORT
    Queue;
  (* A "FIFO.T" is built as an extended "Queue.T".          *)

  TYPE
    T <: Public;

  (* "FIFO.T" is a generic type, which can be instantiated   *
   * using any element type.                                 *)
    Public = FORALL Elem <: ANY IN

             Queue.T(Elem) OBJECT
  (* For any parameter "Elem",                               *
   * "FIFO.Public(Elem)" is an extension of "Queue.T(Elem)". *)

             METHODS
               Push(CONST e: Elem);
             (* The "Push()" method pushes "e" in the queue. *)

               Pop(): Elem;
             (* "Pop()" returns the element at the head of   *
              * the queue and removes it from the queue.     *)
             END;

  END FIFO.
```

(End of Listing 8.9)