Programming Without Enumerations in Oberon

C. Lins
Apple Computer, Inc.
20525 Mariani Ave.
M/S: 37-BD
Cupertino, CA 95014
USA
e-mail (internet): lins@apple.com

When this author attempts to explain the features of Niklaus Wirth's language Oberon [1] to other programmers familiar with Pascal [2] and Modula-2 [3], he has consistently encountered resistance when the listener(s) learn that enumerations are no longer a part of the language. Many programmers have found enumerations useful even without the ability to extend an enumeration across module boundaries — one of the reasons that Niklaus Wirth chose to delete them as a syntactic and semantic construct [1]. Additional reasons given by Professor Wirth for the removal of enumerations included:

- indiscriminate use of enumerations leading to a pompous style contributing to verbosity rather than clarity, and
- specific import rules for enumerations wherein the import of a type caused automatic import of the associated constant identifiers.

In contrast, the C programming language, which lacked enumerations for many years, now has them specified as a part of the ANSI C standard. In addition, C++, Bjarne Stroustrup's object-oriented extension to C, contains enumerations as well [7]. Obviously, there is some disagreement between language designers regarding the utility of enumerations as a specific language feature. The remainder of this article discusses (1) how one would provide the equivalent functionality of enumerations when they are not present as a part of the programming language, (2) how adding enumerations into the syntax of Oberon would needlessly complicate the treatment of array indices, and (3) how in an object-oriented language, enumerations and enumeration constants can be better represented using classes and objects.

Enumerations As Numeric Constants

Without enumerations one must resort to alternative programming techniques for representing a (small) list of conceptually related named constants. Typically, this involves the use of named integer constants (though there is often no reason precluding use of character constants). In the examples that follow we shall presume the existence of a specification calling for codification of a set of result codes for a Channel abstract data type described in [5].

The first representation directly implements the result codes using the named constant concept. The disadvantage of this approach is the assumption that result code values are mutually-exclusive. In other words, only one of the possible result codes occurs following any given channel operation. An Oberon source code fragment implementing the result codes in this manner could be as follows:

```
CONST noError         = 0;
      notDone         = 1;
      tooManyChannels = 2;
      formatError     = 3;
      valueError      = 4;
      useError        = 5;
      lastError       = useError;

VAR   resultCode : INTEGER;
```

Now if we wish to write an error message to the user's display device one might write the following (presuming existence of an input/output library analogous to Modula-2's InOut):

```
CASE resultCode OF
  noError : (* no need to report success *)
| notDone : WriteString("unidentified error");
| tooManyChannels : WriteString("too many channels open");
| formatError : WriteString("input does not conform to the required format");
| valueError : WriteString("input is not representable internally");
| useError : WriteString("requested operation is illegal");
ELSE
  WriteString("unknown result code");
END;
```

Alternatively, one could use the value of the result code as an index into an array containing the appropriate strings to be displayed. Of course, the programmer must explicitly check for result codes outside the array bounds before attempting access of the array. For example, assuming the array has been initialized to the proper values, the array could be declared and accessed with the statements:

```
VAR errorMessage : ARRAY lastError+1 OF ARRAY maxMessageLength OF CHAR;
...
(* errorMessage[noError] is not used and contains the null string *)
IF (noError < resultCode) & (resultCode ≤ lastError) THEN
  WriteString(errorMessage[resultCode]);
ELSIF (resultCode ≠ noError) THEN
  WriteString("unknown result code");
END;
```

One beneficial result of the above encoding of the result codes is the ease with which they are enhanced when there is a new result code. Suppose we have been asked to upgrade the software to support a new device type having an additional result code of "disk full". In this case, the following changes are necessary to our software:

1. add an additional constant declaration `diskFull = 6` to our list of possible result codes;
2. change the constant declaration for `lastError` to read `lastError = diskFull;` and
3. add the appropriate error message string to the initialization statements for the `errorMessage` array.

Another, perhaps less obvious, advantage is that the result codes for error conditions (`notDone` through `lastError`, inclusive) could be hidden in an implementation module. In this way, client module need only be aware of success or failure of the channel operation and simply retrieve the appropriate error message string via a routine exported specifically for this purpose. Thus, client modules would never need be recompiled due to additional error conditions.

This second benefit would not be attainable in Modula-2 or Pascal as the enumeration type defining the possible result codes would usually be declared in a definition module where any change necessitates recompilation of *all* client modules directly importing that altered module. It certainly would not be possible to break apart the enumeration by defining some values in the definition module which are then completed in the implementation module without replacing the use altogether of an enumeration with named constants as shown above.

Since named constants offer the same capabilities as enumerations — without the need for additional language constructs — one can argue that such redundant language features are unnecessary. Certainly they are an additional item of the language feature set that must be (1) learned by programmers becoming familiar the language, (2) taught by instructors, and (3) implemented in the compiler. While some redundancy in a computer programming language is beneficial, excess, unnecessary redundancy is a burden on programmers and compiler writers alike.

The astute reader may have noticed that the visibility problem could be eliminated for enumerations by completely hiding the enumeration within the implementation module and exporting instead a boolean procedure function returning TRUE if an error occurred and FALSE otherwise. In the specific example cited this is true. But the problem persists whenever we wish to extend (or subdivide) an enumeration across module boundaries — precisely one of the reasons for the exclusion of enumerations from Oberon. While there are certain problems with enumerations that in specific situations can be avoided there are other aspects of Oberon where they cannot. This concerns Oberon's array specification mechanism and is the next topic for discussion.

Enumerations and Oberon's Array Mechanism

In Oberon, the array data type is declared following the EBNF grammar:

ARRAY ConstExpr {"," ConstExpr} OF Type

where the constant expressions (ConstExpr) define the *length* of an array in a given dimension. If enumerations were added back into the language, this aspect of the grammar would have to be discarded as a type does not form a constant expression. It would take special rules (specific for enumerations) mapping the number of constant identifiers in an enumeration onto the constant expression describing the length of the array. Such special cases in a language definition are usually a sign of poor language design leading to difficulties for both compiler implementors and programmers alike. Considering the simplicity and benefits attained without the use of enumerations it seems unnecessary to saddle the language with excess syntactic baggage.

Enumerations As Sets

While the above definition for our channel result codes is quite useful as it stands, the mechanism as described suffers from one design flaw — it allows only a single error condition to be active at a given time. For many applications this may be sufficient. But our requirements may specify that multiple error conditions can be active simultaneously. Assuming that our Oberon compiler supports SET types where MAX(SET) is not less than the value for the numeric constant `lastError`, we can easily extend our definitions to account for multiple, simultaneous error conditions. Furthermore, only the implementation need be concerned with the range of possible set values. In this case, the empty set represents the condition of no errors while the non-empty set indicates that *some* error has occurred. The changes to the channel errors module are shown in full in Appendix B.

In Pascal or Modula-2 we could have used a SET with a base type being the enumeration. But the problem of being able to hide the individual enumeration constants while making the set type visible to clients is not possible in either of the aforementioned languages. Only Oberon with its generic SET form allows us this luxury.

Thus, we have seen two mechanisms by which procedurally oriented languages offer alternative forms to that presented by enumerations. As we shall see next, object-oriented languages offer even more powerful opportunities.

Enumerations As Classes and Objects

If we step back for a moment and examine enumerations conceptually for the functionality provided by this language feature we find several interesting results. First, the concept of a type maps closely to the object-oriented concept of a class. Specifically, an enumeration type describes a finite set of possible values which a variable of that type may assume. Such a variable is conceptually equivalent to a singleton set.

Second, classes and their objects encapsulate behavior in the form of *methods*. Often overlooked is that enumerations frequently require additional data structures (and perhaps procedures) for their effective use in an application. For example, an enumeration describing the days of the week in the Gregorian calendar is often cited in elementary programming texts as a suitable use for enumerations; as in

```
TYPE DaysOfWeek = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

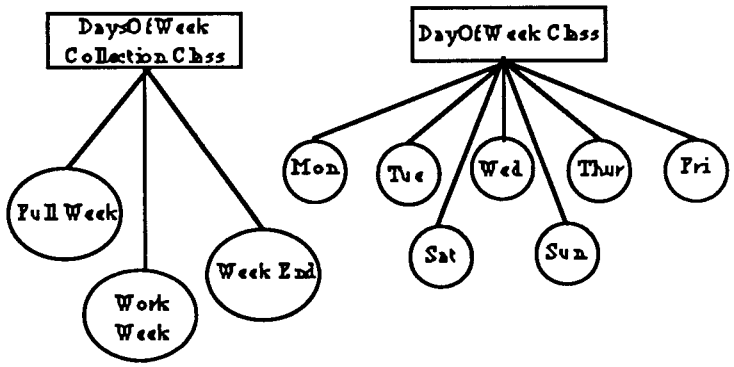
Conceivably, one might desire a textual representation of a day of the week for display to the user. To achieve this end one must generate an additional data structure (e.g., an array of strings indexed by the `DaysOfWeek` enumeration) as well as initialize this structure with the proper values. Unfortunately, the programmer has free rein in placing these types, variables, and procedures. The compiler can in no way enforce these closely related conceptual elements to appear closely related spatially within the text of the program. Such spatial separation of closely related

conceptual elements is clearly detrimental to effective program maintenance — the programmer must search through the source code for the related elements which is an error prone task.

Third, our example program is now clearly tied to the Gregorian calendar. Costly program maintenance will be necessary for conversion of this software to other calendar systems. While this may not be seen as a problem for our example, it is obvious that use of enumerations restricts to varying degrees the portability of software.

Fourth, we cannot use the `DaysOfWeek` enumeration directly for representing alternative weeks (e.g., the work week). For this purpose the programmer must use a different language construct (the subrange) to define such a grouping. Furthermore, this only works due to the specific ordering given in the list of enumeration constants. In actuality, the above enumeration definition not only specifies the names of the days of the week but their ordinal positions relative to one another. Without hardly any work on our part we have created hidden information and dependencies in the program! Clearly, the ease with which hidden dependencies can be introduced when using enumerations indicate a language feature we can do without.

Object-oriented languages allow the programmer to cleanly encapsulate data and behavior via classes and objects. (A very clear short summary of object-oriented concepts appears in [6].) In Object Oberon it is possible to declare a class representing the enumeration type, a class representing an enumeration constant, objects to represent actual enumeration values, and encapsulate all of this is a single module. Thus the principles of information hiding and data abstraction are enforced in the definition of related conceptual program entities. This is depicted below in the following figure where boxes represent classes and circles indicate object instances. The complete module is contained in Appendix C.



From the diagram it can be seen that separation of the day and week concepts greater flexibility is attained. The objects representing the days can be easily grouped arbitrarily and shared by multiple weeks. For example, the week-end is the set of days Saturday and Sunday.

Conclusions

As noted earlier, one of Professor Wirth's complaints against enumerations was indiscriminate use and a verbose style as well as lack of extensibility. It is obvious that named numeric constants, while suffering from the same verbosity problem as enumerations, are not plagued by the extensibility difficulties inherent in the use of enumerations. It is difficult to argue that enumerations should be eliminated as a language feature on the grounds of verbosity. The rationale of eliminating excess redundancy appears to be better grounds for their exclusion from the language.

It has been shown that enumerations, while having a long-standing tradition going back to Pascal, suffer from many hidden problems leading to deterioration in the maintainability of software. Furthermore, alternative language mechanisms exist — especially in object-oriented languages — that are superior to enumerations. Thus, one must conclude that it is probably better to program without enumerations.

Appendix A: Channel Errors Modules (Numeric Constant Version)

```
DEFINITION ChannelErrors;

CONST noError = 0;
VAR   resultCode : INTEGER;

PROCEDURE WriteErrorMessage;
(* Writes an error message to the user's display device depending on the value of
'resultCode'. Nothing is written if 'resultCode' equals noError. Writes the message "unknown
result code" when 'resultCode' is outside the bound of known error conditions. *)

END ChannelErrors.

MODULE ChannelErrors;

CONST maxMessageLength = 45;
      notDone          = 1;
      tooManyChannels  = 2;
      formatError      = 3;
      valueError       = 4;
      useError         = 5;
      lastError        = useError;

VAR errorMessage : ARRAY lastError+1 OF ARRAY maxMessageLength OF CHAR;

PROCEDURE WriteErrorMessage;
BEGIN
  (* errorMessage[noError] is not used and contains the null string *)
  IF (noError < resultCode) & (resultCode ≤ lastError) THEN
    WriteString(errorMessage[resultCode]);
  ELSIF (resultCode ≠ noError) THEN
    WriteString("unknown result code");
  END;
END WriteErrorMessage;

BEGIN
  (*… initialization of errorMessage array …*)
  (* initialize resultCode to 'no error' *)
  resultCode := noError;
END ChannelErrors.
```

Appendix B: Channel Errors Modules (Set Version)

```
DEFINITION ChannelErrors;

CONST noError = {};
VAR   resultCode : SET;

PROCEDURE WriteErrorMessage;
(* Writes an error message to the user's display device depending on the value of
'resultCode'. Nothing is written if 'resultCode' equals the empty set. Otherwise, writes a
comma separated list of error messages with each message corresponding to a specific error
condition that has arisen. *)

END ChannelErrors.

MODULE ChannelErrors;
```

```
CONST maxMessageLength = 45;
      notDone         = 1;
      tooManyChannels = 2;
      formatError     = 3;
      valueError      = 4;
      useError        = 5;
      lastError       = useError;

VAR errorMessage : ARRAY lastError+1 OF ARRAY maxMessageLength OF CHAR;

PROCEDURE WritelErrorMessage (theError : INTEGER);
BEGIN
  (* errorMessage[0] is not used and contains the null string.
     WriteErrorMessage guarantees that theError is a valid index into the
     errorMessage array. *)
  WriteString(errorMessage[theError]);
END WritelErrorMessage;

PROCEDURE WriteErrorMessage;
VAR   index : INTEGER;
BEGIN
  IF (resultCode ≠ noError) THEN
    (* This code deliberately avoids examining potential error codes beyond those already
       defined. *)
    index := notDone;
    WHILE (index ≤ lastError) DO
      IF (index IN resultCode) THEN
        WritelErrorMessage(index);
      END;
      WriteString(", ");
      INC(index);
    END;
  END;
END WriteErrorMessage;

BEGIN
  (*… initialization of errorMessage array …*)
  (* initialize resultCode to 'no error' *)
  resultCode := noError;
END ChannelErrors.
```

Appendix C: DaysOfWeek Modules (Object Version)

```
DEFINITION DaysOfWeekModule;

IMPORT Objects; (* Root of the class hierarchy *)

CLASS Day (Objects.Object);
  PROCEDURE GetTextOf (VAR theText : ARRAY OF CHAR);
    (* return the textual representation for the day *)
  PROCEDURE GetAbbreviationOf (VAR theAbbr : ARRAY OF CHAR);
    (* return the abbreviated textual representation for the day *)
  PROCEDURE Initialize (theText, theAbbr : ARRAY OF CHAR);
    (* initialize the text and abbreviation for the day *)
END Day;

TYPE  DayProcessor = PROCEDURE (Day);
```

```
CLASS Week (Objects.Object);
  PROCEDURE InstallDay (theDay : Day);
    (* add the given day to the week *)
  PROCEDURE GetNthDay  (n : INTEGER) : Day;
    (* return the nth day for the week or NIL if none
       (e.g., NIL for the 6th day of the work week) *)
  PROCEDURE NumDaysInWeek  () : INTEGER;
    (* returns the number of days in this week *)
  PROCEDURE ForAllDaysDo (msg : MESSAGE);
    (* send the message to all days of this week *)
  PROCEDURE Iterate (process : DayProcessor);
    (* iterate over all days of the week *)
END Week;

VAR   fullWeek : Week;
      workWeek : Week;
      weekEnd  : Week;

END DaysOfWeekModule.

MODULE DaysOfWeekModule;

IMPORT Objects; (* Root of the class hierarchy *)
IMPORT Lists; (* Generic, dynamic lists of objects *)

CLASS Day (Objects.Object);
  dayName : ARRAY 11 OF CHAR;
  dayAbbr : ARRAY 4  OF CHAR;

  PROCEDURE GetTextOf (VAR theText : ARRAY OF CHAR);
    BEGIN
      theText := dayName;
    END GetTextOf;

  PROCEDURE GetAbbreviationOf (VAR theAbbr : ARRAY OF CHAR);
    BEGIN
      theAbbr := dayAbbr;
    END GetAbbreviationOf;

  PROCEDURE Initialize (theText, theAbbr : ARRAY OF CHAR);
    BEGIN
      dayname := theText;
      dayAbbr := theAbbr;
    END Initialize;

BEGIN
  (* Clear out any garbage in the strings *)
  dayName := "";
  dayAbbr := "";
END Day;

CLASS Week (Objects.Object);
  members : Lists.List; (* The days in this week - a list of objects *)

  PROCEDURE InstallDay (theDay : Day);
    BEGIN
      members.InsertLast(theDay);
    END InstallDay;

  PROCEDURE GetNthDay  (n : INTEGER) : Day;
    VAR   nthDay : Objects.Object;
    BEGIN
```

```
      nthDay := members.At(n); (* extract the object *)
      RETURN nthDay(Day);      (* Type coerce into a day object *)
    END GetNthDay;

  PROCEDURE NumDaysInWeek  () : INTEGER;
    BEGIN
      RETURN members.LengthOf;
    END NumDaysInWeek;

  PROCEDURE ForAllDaysDo (msg : MESSAGE);
    VAR   index   : INTEGER;
          length  : INTEGER;
          anObject: Objects.Object;
          aDay    : Day;
    BEGIN
      length := members.LengthOf;
      index := 1;
      WHILE (index <= length) DO
        anObject := members.At(index); (* extract day object from the list *)
        aDay := anObject(Day); (* coerce into a day object *)
        IF ACCEPTS(aDay, msg) THEN
          SEND(aDay, msg);
        END;
        INC(index);
      END;
    END ForAllDaysDo;

  PROCEDURE Iterate (process : DayProcessor);
    VAR   index   : INTEGER;
          length  : INTEGER;
          anObject: Objects.Object;
    BEGIN
      length := members.LengthOf;
      index := 1;
      WHILE (index <= length) DO
        anObject := members.At(index);
        process(anObject(Day));
        INC(index);
      END;
    END Iterate;

BEGIN
  NEW(members);  (* Initialize to no members *)
END Week;

(*
AddDayToWeeks creates a day object with the given name and abbreviation, adding the day to
both of the given week objects set of days.
*)

PROCEDURE AddDayToWeeks (    dayName : ARRAY OF CHAR;
                             dayAbbr : ARRAY OF CHAR;
                             week1   : Week;
                             week2   : Week);

VAR   aDay : Day; (* A day object installed in the week's set of days *)

BEGIN
  NEW(aDay);
  aDay.Initialize(dayName, dayAbbr);
  week1.InstallDay(aDay);
  week2.InstallDay(aDay);
```

```
END AddDayToWeeks;

BEGIN
  (* Initialize the whole module by declaring the standard week objects and
     initializing their sets of days *)
  NEW(fullWeek);
  NEW(workWeek);
  NEW(weekEnd);
  AddDayToWeeks("Sunday", "Sun", fullWeek, weekEnd);
  AddDayToWeeks("Monday", "Mon", fullWeek, workWeek);
  AddDayToWeeks("Tuesday", "Tue", fullWeek, workWeek);
  AddDayToWeeks("Wednesday", "Wed", fullWeek, workWeek);
  AddDayToWeeks("Thursday", "Thur", fullWeek, workWeek);
  AddDayToWeeks("Friday", "Fri", fullWeek, workWeek);
  AddDayToWeeks("Saturday", "Sat", fullWeek, weekEnd);
END DaysOfWeekModule.
```

References

[1] N. Wirth, "The Programming Language Oberon," *Software Practice and Experience*, 18 (7), (July 1988), 671-690.

[2] N. Wirth and K. Jensen, *Pascal User Manual and Report*, 2nd edition, Springer-Verlag, New York, 1974.

[3] N. Wirth, *Programming in Modula-2*, 4th edition, Springer-Verlag, Berlin, 1988.

[4] N. Wirth, "From Modula to Oberon," *Software Practice and Experience*, 18 (7), (July 1988), 661-670.

[5] M. Odersky, "MINOS: A New Approach to the Design of an Input/Output Library for Modula-2," *Structured Programming*, 10 (2), 1989, 89-105.

[6] G. Blaschek, "Implementation of Objects in Modula-2", *Structured Programming*, 10 (3), 1989, 147-155.

[7] S. B. Lippman, *C++ Primer*, Addison-Wesley, Reading, MA, 1989.