Another defence of enumerated types

Markku Sakkinen †
Department of Computer Science and Information Systems
University of Jyväskylä
PL 35, SF-40351 Jyväskylä, Finland
Internet: sakkinen@jytko.jyu.fi Bitnet/EARN: SAKKINEN@FINJYU

Abstract

I claim that enumerations, while of course not strictly necessary, are an elegant and useful facility in modern programming languages. I try to show that arguments recently given against them are weak at best and bogus at worst, for general-purpose programming. Some related issues on types in programming languages are touched as well. These make it even more questionable whether Oberon marks progress or regress in language design.

1. Introduction

The debate on enumerations in the February 1991 issue between Mark Cashman [7] and Charles Lins [8] was interesting. It prompted me to look up Lins' original article [6], which I had not read before. Lins' response to Cashman's article seems (naturally enough) to be written in a haste, and thus it suffers from more inaccuracies itself than Lins charges Cashman with — I hope this comment of mine is not worse still. Most importantly, as the title already tells, Cashman's paper is a defence of enumerations in general, not only against those arguments espoused by Lins.

An early commentator on Oberon [4] flatly stated:

> I fully agree with the elimination of these superfluous and error-sensitive facilities [...]

(p. 23), meaning all Modula-2 features eliminated from Oberon except for a couple of van Delft's own favourites. However, the article gave no arguments on why he (she?) considered any of these features (including opaque types) "superfluous and error-sensitive".

Let me begin with personal reminiscences, so you see on what side my heart is. Some ten years ago, when I was doing "real work" (cf. current affiliation), the possibility to switch from Fortran to Pascal felt wonderful. One of the most important advantages were enumerations, which allowed me to write much clearer and more self-documenting code. Very probably the first enthusiasm led me even to overuse them.

When I cursorily read about Oberon [1,2], my general reaction was: now Wirth tries to banish, as far as possible, every feature that is known to be misused by some programmers. To me it seemed like taking away all sharp knives from a surgeon so that (s)he may not hurt her/his fingers. Still, if enumerations were Wirth's own original invention (I am not an expert in the history of programming languages), it is quite remarkable that he has had the heart to abandon them.

Cashman in turn writes as his final conclusion:

> Oberon has some features of interest, but it does not seem to be a major advance over Modula-2. Indeed, in some areas of type-safety, it seems to be a step backward.

While largely agreeing, I suspect that Cashman had not studied Oberon except from [6], so it might have been wise to qualify his judgement more clearly. In one place, he complains about the Modula-2 principles of partial import ([7], p. 36) and suggests that explicit qualifiers should always be used with imported names. Oberon in fact requires this explicit qualification.

Two predecessors of Oberon from the drawing board of Niklaus Wirth, Pascal and Modula-2, have become very popular. It is interesting to compare Oberon with Modula-3 [5], another descendant of Modula-2, which has been developed with Wirth's blessing and advice. In contrast to Oberon, Modula-3 has retained enumeration types; it is even in general less Spartan than Oberon.

In the following section, I try to argue about enumerations and some related issues from a somewhat philosophic point of view. In the remaining sections, we will look again at the main points treated by Lins and Cashman. We may skip Wirth's argument about verbosity [1]; Lins indeed did not support it in [6], although Cashman claims so.

I am not an expert in Modula-2, Modula-3 nor Oberon, and the available literature does not always seem to give unambiguous answers to all questions. Therefore some technical details in the sequel may not be completely accurate (v. Acknowledgements), but that should not distract from the principles.

2. Why have so many types?

The original purpose of Oberon [1] clearly was to be a compact and elegant language for building compact and elegant operating systems and the like. The design choices made in the language may be very appropriate for this purpose. However, at least Lins [6]

† Work supported by the Academy of Finland, Project 1061120 (Object-oriented languages and techniques).

seems to advocate Oberon for general-purpose programming, where the tradeoffs can be quite different.

From some viewpoint, every distinct datatype that appears in a piece of software is a liability; for instance because it may make reasoning about the software more complicated. From a *modelling* viewpoint [3], however, it is desirable for types and entities in the software to correspond as closely as possible to things perceived in the real world. This viewpoint is important in large applications, especially when their specifications are fuzzy and change rapidly.

Subrange types are another traditional feature that has been eliminated from Oberon (but retained in Modula-3). I do not mourn for them so much: subranges have weaker advantages and create greater problems than enumerations. Especially, there is often need for arithmetic and conversions between instances of different subranges of the same base type.

For the modelling power and convenience of the languages, it is a retrograde step in Modula-3 that *type equivalence* is defined as structural equivalence[1], while Modula-2 applies "name equivalence" (the most common although misleading term; something like 'declaration identity' might be better). The possibility to make similar types distinct prevents programmers from accidentally adding apples to oranges.

The principle of structural type equivalence would be worse in Oberon than in Modula-3. because it has also abandoned opaque types. Some details in [2] (especially the definition of procedure types) would imply that structural equivalence was meant, some others (especially the principle of record type extension) again hint to the converse. It is surprising that Wirth has again eschewed this important issue[2]: it caused a lot of problems and debate with Pascal in the 70's. At least current Oberon implementations seem to apply name equivalence (v. Acknowledgements).

Enumerated types are one feature that helps programmers to keep unrelated entities clearly separated. The advantages of well-chosen enumerations really become apparent only when there are several of them in the same programme; that could not be demonstrated in a short article like [7]. One must bear in mind that the utility of many a language feature looks very different in toy examples on one hand and life-size software on the other hand.

Enumerations have some similarity in principle with the *classes* of object-oriented programming ('object types' in Modula-3). Both allow programmers to define types and entities that correspond much more closely to problem-domain concepts than do the built-in types, which are more related to the facilities of computer hardware. With classes, the programmer can define new *complex* types with arbitrary semantics; with enumerations, new *primitive* types with very restricted semantics.

3. Enumerations vs. classes

Lins ([6], p. 21 – 22) essentially seems to mean that in object-oriented languages, classes make enumerations redundant or obsolete. I beg to differ from this opinion, referring to the previous section.

A class (or a Modula-2 opaque type or an Ada *private* type) differs from conventional record or structure types in that the operations that can be applied to its instances are restricted to what the class designer has considered semantically meaningful. Clients cannot arbitrarily inspect or modify the instance variables of an object. Similarly, enumerations differ from integers in that arithmetic operations, which would have no sensible meaning when applied to a classification, are prohibited.

Certainly, the semantics of enumerations in Modula-2 and other current languages do not fit all desirable purposes exactly (although better than integers). For instance, there are classifications with no inherent order between the items, e.g. the channel errors in [6, 7]. There are also cases in which the natural order is cyclic, e.g. the days of the week. I could somehow imagine a language facility of "enumeration classes" for the exact tailoring of operations and their semantics, but its advantages would probably not be worth the added complexity.

A general facility for the definition of (possibly many-sorted) algebras of atomic values would subsume enumerations, of course. Ada actually seems to have sufficient features for this; thus enumerations seem genuinely redundant (but convenient) in Ada, but not so in any of the three languages that we are primarily discussing.

Although I am a proponent of object-oriented programming myself, I agree with Cashman that in the presentation of [6], classes do not yet introduce anything essential into the weekday example that goes beyond the ordinary facilities of Modula-2, in spite of the great length of Appendix C. In fact, many object-oriented languages (obviously not Object Oberon) are decisively weaker than Modula-2 and Ada in the respect that they lack modules or packages: the only available unit of modularisation is a single class.

Some of the arguments given on p. 21 – 22 of [6] against the declaration of the days of the week as an enumeration, frankly, look ludicrous to . me. For instance the facts that the week has 7 days and Tuesday follows Monday exist in the problem domain, instead of being "hidden information and dependencies"

[1] The attribute 'branded' allows name equivalence to be defined, but only for reference types.

[2] A revised version of the report, dated 1 October 1990 and available by anonymous FTP from ETH (Eidgenössische Technische Hochschule = Swiss Federal Institute of Technology, Zurich) gives no more information on this point.

created by the programmer.

4. Language-specific flaws of Modula-2

There are some deficiencies in the treatment of enumeration types in Modula-2; these become very evident in comparison to Ada, which has done the things right. If Wirth's goal with Oberon had not been an almost minimal language, he might have corrected these flaws instead of omitting enumerations altogether.

First of all, an enumeration cannot be exported opaquely. This restriction cannot be defended even on the grounds of implementation difficulties, since subranges of standard types *can* be opaquely exported. I would suggest it to be lifted, more strongly than Cashman on p. 36 of [7]. — Ada allows *any* type to be declared *private* in a package specification, which is already more demanding on implementors. In Modula-3, only reference types may be declared opaque ([5], p. 31).

One of the arguments of Wirth ([1], p. 663) that Lins cites is the following, closely related to opacity:

> ... the exceptional rule that the import of a type identifier also causes the (automatic) import of all associated constant identifiers.

Why not add an option to the IMPORT declaration so that the import of enumeration constants can be prevented or made selective? — This possibility does not exist in Ada either. Almost the same effect can be achieved by using a private type together with the following feature.

Named constants in Modula-2 and Oberon can be of standard types only: an obvious unorthogonality between constants and variables. In Ada and Modula-3 named constants of user-defined types can be declared. Furthermore, such a declaration in the *specification* part of an Ada package need not contain the value of the constant. Therefore, one can get the effect of a selective export of enumeration constants in Ada by making the type itself private and declaring the desired named constants.

The "name space pollution" and name conflicts that can be caused when a large number of enumerations are visible within the same scope can be controlled with proper language mechanisms. In Ada, enumeration constants can be qualified by the type name if they would be ambiguous otherwise, e.g.

```
Weekday'(Sunday)
```

It would seem ([5], p. 3 – 4) that Modula-3 *always* requires such qualification:

```
Weekday.Sunday
```

The inability to have an array or record as the result type of a function procedure was one of those complaints in [7] (p. 38) not directly connected with the main theme. Although defendable on a cost–benefit basis, it is an ugly unorthogonality in Modula-2 and Oberon; it seems to be corrected in Modula-3. Since 'set' is a basic type in Oberon (§7), it is allowed as the return type of a function; in Modula-2 set types are constructed types and suffer from the same restriction as array and record types.

5. Enumerations vs. numeric constants

In the first example, on channel error codes, Cashman's first solution with enumerations would become in almost all respects superior to Lins' solution with named integer constants, if only we could make ChannelErrorTYPE opaque (§4). (Thus, it would be possible in Ada.) I will not repeat the arguments given in [7], p. 35 – 36. It could be useful to add a procedure like

```
PROCEDURE ChannelError
    (Code: ChannelErrorTYPE): BOOLEAN;
```

An additional benefit of this approach will be seen in §7.

Lins writes in [8] about Cashman's second solution:

> ... he explicitly avoids the use of enumerations, instead using an opaque type. Here he has given an excellent example of how *not* using enumerations yields a more elegant, and possibly extensible, solution.

Lins has perhaps misunderstood Cashman's *purpose*: the enumeration has not been omitted, but only moved from the definition to the implementation, to achieve information hiding. This possibility had been admitted in [6], p. 20. However, even I really cannot see much use for the enumeration in this example.

Enumerations automatically avoid the "apples to oranges" problem mentioned in §2. With integers, nothing warns us about assigning Wednesday to a variable that was supposed to contain a channel error code. In Modula-2, if we wanted to avoid enumerations, we could make *variables* representing error codes and days of the week mutually incompatible by defining

```
TYPE ChannelErrorTYPE = INTEGER;
     DayTYPE = INTEGER;
```

This would evidently work as intended also in Oberon (at least in the ETH implementation), but not in Modula-3 because of its principle of structural type equivalence. Unfortunately, in Modula-2 and Oberon there is no way to declare a named *constant* to be of type ChannelErrorTYPE or DayTYPE (§4).

Incidentally, Cashman's second example also suffers from the just-mentioned deficiency of Modula-2, He declares:

```
VAR HandleNIL: HandleTYPE;
PROCEDURE Open
    (ChannelName: ARRAY OF CHAR;
     VAR ErrorString: ARRAY OF CHAR):
  HandleTYPE
    (* Returns HandleNIL on fatal error *);
```

It was not possible to declare HandleNIL as a constant, which it logically should be. Now clients can modify HandleNIL inadvertently. Since the only purpose of exporting HandleNIL is to compare HandleTYPE return codes to it, I would rather replace it with a procedure, as at the beginning of this section:

```
PROCEDURE ChannelError
  (Handle: HandleTYPE): BOOLEAN;
```

The argument that the integer solution is more easily extensible, without needing recompilations, is double-edged at best. Namely, there is not much sense in adding a new code unless it is taken into account in all procedures that treat error codes. This might be more easily forgotten with integers than with an enumeration.

The recompilation work caused by modified enumeration definitions should not be overemphasised. Compare this to the C++ language: almost any modification in a class, even its private part, requires all subclasses and all client files of that class to be recompiled. Nevertheless C++ seems to be very popular.

6. Enumerations as array indices

In Oberon, each dimension of an array is declared by giving the length as a constant expression. Lins says in [6], p. 21, that enumerations as index types (dimensions) would not fit nicely into this model. That is true but irrelevant. The Oberon convention is no better than using index types (as in Pascal) or bound pairs (as in so many languages since Algol 60), both of which are suitable for enumerations.

Going back to memories (§1), I would say that the index types of arrays in the industrial automation software written by my team were mostly enumerations. I would also guess that most 'case' statements were based on enumerations.

7. Sets of enumerations

Lins writes in [6], p. 21:

> ... being able to hide the individual enumeration constants while making the set type visible to clients is not possible in [Pascal or Modula-2].

In Ada, even a set type can be private (§4); if it is not declared *limited* private, clients can assign and compare set values. On rare occasions, I admit, there might be some advantage in having a set type visible without knowing anything about its base type, so that one can apply standard set operations.

We should note that the error code example presented by Lins is very peculiar, although not uncommon. In the first case of a single return code, the alleged advantage of the integer representation depends totally on the circumstance that there is only one code that signifies success, and its representation is known (to be 0). In the second case where a set of codes is returned, the advantage similarly depends on the fact that all status codes in the set signify errors, and thus the empty set has a special meaning.

If the situation is even a bit more general, we will need two parallel return values if we do not want to reveal everything to clients: one that tells whether the operation succeeded or failed, and another (opaque) that contains the details. — Alternatively, as in the suggestions of §5, we may simply keep the return code type opaque and provide a Boolean procedure to tell about success or failure. We already get a clear advantage over the solution with integer constants: if we decide to switch from a single code to a set or vice versa, client code need not be modified.

Lins regards it as an *advantage* of Oberon that it has only one general set type, as opposed to distinct set types for each base type. In my opinion, this again reduces the modelling power of the language. However, it is a rather necessary consequence from the fact that both enumerations and subranges have been omitted from the language — there would not be many potential base types anyway.

8. Output and input of enumerations

Lins writes on p. 21 of [6]:

> Conceivably, one might desire a textual representation of a day of the week for display to the user.

He then goes on to explain why he thinks this to be difficult with enumerations (I was not convinced). However, when Cashman suggests on p. 38 of [7]:

> There should be a string which is the identifier in string form, associated with each enumeration element. An intrinsic function could make the string available as needed.

Lins retorts in [8]:

> While perhaps of marginal use to a programmer, exposing a programming identifier to an end-user as part of an error message is clearly undesirable.

In many cases (§2), an enumeration exactly models a problem-domain classification, and the identifiers can be chosen so that even an end user can relate to them. The days of the week are a prime example. Certainly there are many situations in which outputting an enumeration identifier could only confuse the user; but likewise most numeric variables in a programme are such that there would be no use to print out their values, and still nobody wants to prohibit the output of integers in general.

Multilingual environments create additional problems, but those are not insurmountable. It would be possible to check at compile or link time that strings corresponding to all enumeration constants have been defined for all alternative user languages to be supported. Indeed such checking would appear much easier implementable for enumerations than other approaches.

There are needs for the *input* of enumerations as well, although probably less often than output. Since the input is far less trivial to program by hand in a robust way than the output, it could be even more beneficial to have as a built-in feature. I think some Pascal implementations already have facilities for the input and output of enumerations.

9. Conclusion

It is true that omitting enumerations from a programming language makes the compiler writer's job easier. Therefore it may indirectly profit even programmers, because the compilers may be smaller, faster, and less bug-ridden. However, I suppose enumerations are not a particularly difficult feature to implement. Leaving out integer types would bring far greater savings; after all, everything *can* be done with just floating-point numbers.

Lins (and Wirth) have tried to convince us that dispensing with enumerations is a direct advantage also to the users (programmers) of a language. They should be first-rate experts; Lins has published books about software components in Modula-2. However, many of their arguments turned out to be evidence of flaws in Modula-2, not in the general idea of enumerated types. Although bashing Ada seems to be fashionable, language designers might sometimes do well to look for well-thought ideas and features in Ada.

Until much stronger arguments than we have seen so far are presented, I agree with Cashman: one *can* do without enumerations, but at the expense of more work and less understandable, less maintainable code. One aspect that Cashman did not take up explicitly but that I want to stress is that judicious use of enumerations is an aid in the modelling of the application domain.

Acknowledgements

When I had submitted the previous version of this article and sent copies to some acquaintances, Kai Koskimies told me that at least the ETH implementation of Oberon is based on "name equivalence" between types. When I asked on Usenet about type equivalence and the restrictions on function return types, Thomas Römke and Andreas Borchert pointed out that I had missed the restriction (§4) that was clearly written in §10.1 of [2]. Marc-Michael Brandis then confirmed (in the 'comp.lang.modula2' group):

> Oberon uses name equivalence as does Modula-2. There are two special cases in which structural equivalence is used: Open arrays and procedure variables.

I would also like to thank the editor for his patience with revisions. The first version of this paper was already patched once with replacement pages because of some erroneous reference numbers.

References

These are in chronologic order. Since there are any number of books on both Modula-2 and Ada, references to them would be superfluous.

[1] Niklaus Wirth. "From Modula to Oberon". *Software — Practice and Experience,* 18:7 (July 1988), 661 – 670.

[2] Niklaus Wirth. "The Programming Language Oberon". *Software — Practice and Experience,* 18:7 (July 1988), 671 – 690.

[3] Ole Lehrmann Madsen, Birger Møller-Pedersen. "What object-oriented programming may be – and what it does not have to be". In: S. Gjessing, K. Nygaard (Eds.). *ECOOP '88 European Conference on Object-Oriented Programming,* 1 – 20. Springer-Verlag 1988.

[4] A.J.E. van Delft. "Comments on Oberon". *ACM SIGPLAN Notices,* 24:3 (March 1989), 23 – 30.

[5] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. *Modula-3 Report (revised).* Digital Systems Research Center and Olivetti Research Center, 1989.

[6] C. Lins. "Programming Without Enumerations in Oberon". *ACM SIGPLAN Notices,* 25:7 (July 1990), 19 – 27.

[7] Mark Cashman. "The Benefits Of Enumerated Types in Modula-2". *ACM SIGPLAN Notices,* 26:2 (February 1991), 35 – 39.

[8] C. Lins. Author's Response to [7]. *ACM SIGPLAN Notices,* 26:2 (February 1991), 40.