# Differences between Oberon and Oberon–2

H. Mössenböck, N. Wirth
Institut für Computersysteme, ETH Zürich

Oberon–2 is a true extension of Oberon [1]. This paper summarizes the extensions and tries to shed some light on the motivations behind them. By that we hope to make it easier for the reader to classify Oberon–2. For details the reader is referred to the language report.

One important goal for Oberon–2 was to make object–oriented programming easier without sacrificing the conceptual simplicity of Oberon. After three years of using Oberon and its experimental offspring Object Oberon [2] we merged our experiences into a single refined version of Oberon.

The new features of Oberon–2 are type–bound procedures, read–only export of variables and record fields, open arrays as pointer base types, and a with statement with variants. The for statement is reintroduced after having been eliminated in the step from Modula–2 to Oberon.

Oberon–2 is the result of many discussions among all members of the Institute for Computer Systems at ETH. It is particularly influenced by the ideas of Jürg Gutknecht and Josef Templ.

### Type–bound procedures

Procedures can be bound to a record (or a pointer) type. They are equivalent to methods in object–oriented terminology. The binding is expressed by a separate parameter (the operand to which the procedure is applicable, or the "receiver" as it is called in object–oriented terminology).

```
TYPE
    Figure = POINTER TO FigureDesc;
    FigureDesc = RECORD
        x, y, w, h: INTEGER
    END;

PROCEDURE (f: Figure) Draw; BEGIN ...  END Draw;
PROCEDURE (f: Figure) Move (dx, dy: INTEGER); BEGIN ...  END Move;
```

*Draw* and *Move* are bound to *Figure* which means that they are operations applicable to *Figure* objects. They are considered local to *FigureDesc* and can be referenced like record fields, e.g. *f.Move(10, 10*) if *f* is a variable of type *Figure*.

Any procedure bound to a type *T* is implicitly also bound to all extensions of *T*. It can be redefined (overridden) by a procedure with the same name and the same formal parameter list which is explicitly bound to an extension of *T*, such as in

```
TYPE
    Circle = POINTER TO CircleDesc;
    CircleDesc = RECORD (FigureDesc)
        radius: INTEGER
    END;

PROCEDURE (c: Circle) Move (dx, dy: INTEGER);
BEGIN
END Move;
```

*Circle* is an extension of *Figure*. A procedure *Move* is explicitly bound to *Circle* and redefines the *Move* that is "inherited" from *Figure*. Let *f* be a variable of type *Figure* and *c* a variable of type *Circle*; then the assignment *f := c* makes the dynamic type of *f* (its run time type) be *Circle* instead of *Figure*. In the call

```
f.Move(10, 10)
```

the variable *f* serves two purposes: First it is passed as the receiver parameter to the procedure *Move*. Second, its dynamic type determines which variant of *Move* is called. Since after the assignment *f := c* the dynamic type of *f* is *Circle*, the *Move* that is bound to *Circle* is called and not the one that is bound to *Figure*. This mechanism is called dynamic binding, since the dynamic type of the receiver is used to bind the procedure name to the actual procedure.

Within a redefining procedure the redefined procedure can be invoked by calling it with the suffix ^, e.g. *f.Move^ (dx, dy)*.

*Motivation*. We refrained from introducing the concept of a class but rather replaced it by the well–known concept of records. Classes are simply record types with procedures bound to them.

We also refrained from duplicating the headers of bound procedures in the record as it is done in other object–oriented languages like C++ or Object Pascal. This keeps record declarations short and avoids redundancy (changes to a header would have to be made at two places in the program and the compiler would have to check the equality of the headers). If the programmer wants to see the record together with all procedures bound to it he uses a tool (a browser) to obtain the information on screen or on paper.

The procedures bound to a type may be declared in any order. They can even be mixed with procedures bound to a different type. In Object Oberon, where all methods have to be declared within their class declaration, it turned out that indirect recursion between methods of different classes make awkward forward declarations of whole classes necessary.

In languages like Object Pascal or C++, instance variables of the receiver object *self* can be accessed with or without qualification (i.e. one can write either *x* or *self.x*). In these languages it is sometimes difficult to see whether a name is an ordinary variable or an instance variable. It is even more confusing if the name denotes an instance variable that is inherited from a base class. We therefore decided that instance variables must always be qualified in Oberon–2. This avoids having a choice between two semantically equivalent constructs, which we consider undesirable in programming languages.

In Oberon–2, the receiver is an explicit parameter, so the programmer can choose a meaningful name for it, which is usually more expressive than the predeclared name self that is used in other object–oriented languages. The explicit declaration of the receiver makes clear that the object to which an operation is applied is passed as a parameter to that operation. This is usually not expressed in other object–oriented languages. It is in the spirit of Oberon to avoid hidden mechanisms.

In Object Oberon methods have the same syntax as ordinary procedures. In large classes where the class header is not visible near the method header it is impossible to see whether the procedure is an ordinary procedure or a method, and to which class the method belongs. In Oberon–2, the type of the receiver parameter of a bound procedure denotes the type to which the procedure is bound, so no confusion can arise.

**Read–only export**

While in Oberon all exported variables and record fields can be modified by a client module, it is possible in Oberon–2 to restrict the use of an exported variable or record field to read–only access. This is expressed by marking its declaration with a "–" instead of a "*". The "–" suggests the restricted use of such a variable.

```
TYPE
   Rec* = RECORD
      f0*: INTEGER;
      f1–: INTEGER;
      f2: INTEGER
   END;

VAR
   a*: INTEGER;
   b–: Rec;
   c: INTEGER;
```

Client modules can read the variables *a* and *b* as well as the fields *f0* and *f1*, since these objects are exported. However, they can modify only *a* and *f0*; the value of *b* and *f1* can be read but not modified. Only the module which exports these objects can modify their values. (Even if clients declare a private variable of type *Rec*, its field *f1* is read–only.) Since *b* is read–only, its components are read–only, too.

The motivation behind read–only export is to allow a finer grain of information hiding. Information hiding serves two purposes: First, it helps to keep off unnecessary details from clients. Second, it allows establishing the assertion that the values of hidden variables are only modified by access procedures of the module itself, which is important to guarantee invariants. Read–only export supports the second goal.

**Open arrays**

Both in Modula–2 and in Oberon it is possible to have open arrays as parameters. The length of such an array is given by the length of the actual parameter. In Oberon–2 open arrays may not only be declared as formal parameter types but also as pointer base types. In this case, the predeclared procedure NEW is used to allocate the open array with arbitrary length.

```
VAR v: POINTER TO ARRAY OF INTEGER;
... NEW (v, 100)
```

The array *v^* is allocated at run time with a length of 100 elements accessed as *v[0]* to *v[99]*.

**With statements**

In Oberon, a with statement is a regional type guard of the form

```
WITH v: T DO S END
```

If the variable *v* is of dynamic type *T*, then the statement sequence *S* is executed where a type guard *v(T)* is applied to every occurrence of *v*, i.e. *v* is regarded as if it had the static type *T*. If the dynamic type of *v* is not *T* the program is aborted. In Oberon–2, the with statement can be written with variants, e.g:

```
WITH v : T0 DO S0
|  v : T1 DO S1
ELSE S2
END
```

If the dynamic type of *v* is *T0*, then *S0* is executed and *v* is regarded as if it had the static type *T0*; if the dynamic type of *v* is *T1*, then *S1* is executed and *v* is regarded as if it had the static type *T1*; else *S2* is executed. If no variant can be executed and if an else clause is missing the program is aborted.

**For statements**

Although for statements can always be expressed by while statements, they are sometimes more convenient because they are shorter and termination is inherent. This is the case if the number of iterations is fixed like in many applications dealing with arrays. The for statement is written as:

```
FOR i := a TO b BY step DO statements END
```

This statement is equivalent to the statement sequence

```
temp := b; i := a;
IF step > 0 THEN
    WHILE i <= temp DO statements; i := i + step END
ELSE
    WHILE i >= temp DO statements; i := i + step END
END
```

References

[1] N.Wirth: The Programming Language Oberon. Software Practice & Experience, 18 (1988)
[2] H.Mössenböck, J. Templ: Object Oberon – A Modest Object–Oriented Language.
Structured Programming 10/4: 199–207, 1989