Object Oberon
An object-oriented extension of Oberon

Eidgenössische
Technische Hochschule
Zürich

Department Informatik
Institut für
Computersysteme

H. Mössenböck
J. Templ
R. Griesemer

Object Oberon

An Object-Oriented Extension of Oberon

June 1989

109

Authors’ address:

Institut für Computersysteme
ETH-Zentrum
CH-8092 Zürich, Switzerland

e-mail: moessenboeck@inf.ethz.ch

Abstract

Object Oberon extends Oberon [1] by the concept of classes, making object-oriented programming in Oberon more natural. The added features leave Oberon almost unchanged and cause only a negligible overhead for method calls in object code and at run time.

Contents

1. Motivation

The language Oberon [1] provides the concept of type extension, which is a prerequisite to object-oriented programming. Oberon itself is intentionally not a fully object-oriented language, i.e. it does not allow the declaration of classes with methods that can be overridden in subclasses (known as *dynamic binding*). This report describes a small language extension of Oberon which adds the concept of classes. The reason for the extension was the need for a simple language that could be used in a course on object-oriented programming at ETH. Our aim was to retain Oberon as the base language and to make the changes to it as small as possible. Another goal was to define a language which supports object-oriented programming in a minimal way. Thus, only essential features were included.

In order to demonstrate the need for a more convenient language mechanism, we first show how object-oriented programming can be done in conventional Oberon. We then define the language extension. Finally we sketch the implementation of classes and give some measurements. Appendix 1 summarizes the Object Oberon grammar and Appendix 3 describes a basic class library written in Object Oberon.

2. Object-Oriented Programming in Oberon

There are various possibilities for simulating and object-oriented programming style in Oberon, although so far none of them have been described explicitly. the Oberon report does not even mention the term “object-oriented programming”.

The concepts of object-oriented programming correspond to the following Oberon concepts: *Classes* are records and *objects* are variables of a record type; *instance variables* are record fields and *methods* are procedure variables declared as record fields; *inheritance* between classes corresponds to type extension of records; *superclasses* are base types and *subclasses* extensions of a base type; *sending a message* simply means calling the corresponding procedure variable.

Basically, there are two ways to use Oberon for object-oriented programming: (1) All methods are declared as procedure variables in records; (2) every class has a single procedure variable (a handler) which receives and distributes messages sent to the object.

2.1 Classes with Procedure Variables as Methods

Using ordinary procedures to access the data in a record is sometimes unsatisfactory, since such procedures are called explicitly and cannot be overridden for a derived type. Therefore, such systems are not really extensible. A better way is to implement the operations as procedure variables and to include them in the record so that new procedures can be installed in their place for every derived type. For example, a stack class can be implemented as follows:

```
Stack = POINTER TO StackRec;
StackRec = RECORD
    s: ARRAY 128 OF INTEGER;
    sp: INTEGER;
    Push: PROCEDURE (Stack, INTEGER);
    Pop: PROCEDURE (Stack): INTEGER;
END;
```

This is a natural way to implement classes in Oberon, but it has several disadvantages:

- The methods take up space in every object. In deep class hierarchies 20 or more methods are not unusual.
- The procedure variables have to be initialized every time a new object is created. This is tedious and error prone.
- The receiver of a message (the object whose data is to be manipulated by a method call has to be passed as a parameter of every method in order to make the private data on the object accessible.

Experience has shown that this sort of object-oriented programming tends to make things harder rather than easier.

2.2 Classes with a Message Handler

The following is how classes are implemented in the Oberon system [2], thus we regard it as the recommended way to write object-oriented programs in Oberon: Every class has a single procedure variable called its *handler,* which receives messages consisting of an identification number and some message parameters. Using the message identification, the handler takes appropriate action (usually it calls some procedure). The stack class example turns into:

```
Message = RECORD id: INTEGER; val: INTEGER END;
Stack = POINTER TO StackDesc;
StackDesc = RECORD
    s: ARRAY 128 OF INTEGER;
    sp: INTEGER;
    handle: PROCEDURE (Stack, VAR Message)
END;
```

Here, the handler would be sketched like:

```
PROCEDURE *Handle (stack: Stack; VAR msg: Message);
BEGIN
    CASE msg.id OF
        0: INC(stack^.sp); stack^.s[stack^.sp] := msg.val       (* message "Push" *)
    |   1:msg.val := stack^.s[stack^.sp]; DEC(stack^.sp)       (* message "Pop" *)
    END
END Handle;
```

This scheme is flexible, since new messages can be added to a class without changing the interface to clients. (A new message is simply a new identification number.) But it has some drawbacks, too:

- It is slow because the handler has to distinguish between the messages at run time. Every method call involves a message lookup by the handler.
- It is difficult to read, since a class declaration gives no hints which messages can be sent to an object and what parameters are allowed.
- The handler must be installed explicitly for every object created.
- The receiving object still has to be passed as a parameter of a message.
- Sending a message is very clumsy: it involves setting up the message record (identification and other message parameters) and calling the handler.

The scheme works well when there are few objects, and messages are sent rather infrequently. But it is inappropriate when programs are written in an object-oriented style where objects and methods are the basic building blocks of a program.

3. The Language Object Oberon

Object Oberon is based on Oberon adding to it the concept of classes. Oberon has been designed for writing extensible and efficient systems. This is also the aim of Object Oberon: classes help to make systems yet more extensible, and as we will show in Chapter 5, they can be implemented to be more efficient than the schemes described in the previous chapter.

This chapter describes how classes are embedded into Oberon. It is not a full language report. Everything which is left out here is defined in the same way as it is in the Oberon report.

3.1 Lexical Additions

The reserved words of Oberon are augmented by the keyword CLASS. Comments may be nested.

3.2 Classes

A class is an abstract data type consisting of private data and operations defined on this data.

$ ClassDeclaration = CLASS $ident_1$ [SuperClass] ";" Body END $ident_2$.
$ Body = Fields Methods [BEGIN StatementSequence].

$ident_1$ is the class name and must match *$ident_2$*. A class constitutes a scope for all names declared in *Body*. The visibility rules for names declared in a class scope are the same as for names declared in a record scope. The statement sequence in *Body* is executed whenever an object of this class is allocated (see 3.3). It can be used to initialize the instance variables of an object.

$ SuperClass = "(" qualident ")".

A class C may be derived from another class C0 specified by *qualident*. In the declaration

```
CLASS C (C0); ... END C;
```

C0 is called the direct superclass of C and C a direct subclass of C0. The term superclass (subclass) means the direct superclass (subclass) or a superclass (subclass) thereof. A subclass inherits all instance variables and methods from its superclass(es).

$ Fields = FieldListSequence.
$ Methods = {ProcedureDeclaration ";"}.

Fields (also referred to as instance variables) are equivalent to record fields and methods are equivalent to procedure declarations in Oberon.

Example:

```
CLASS Viewer(Frame);
    mode: INTEGER;
    visible: BOOLEAN;

    PROCEDURE Open(x, y: INTEGER; mode: INTEGER); BEGIN . . . END Open;
    PROCEDURE Close; BEGIN ... END Close;
    PROCEDURE Move(x, y: INTEGER); BEGIN . . . END Move;

BEGIN visible := FALSE
END Viewer;
```

3.3 Objects

Objects are variables whose type is a class. If an object *v* is of type *Viewer,* its fields can be accessed by v.mode and v.visible. Its methods can be called by v.Open(100, 100, 0), etc. Both fields and methods may be used without qualification within the class that declares or inherits them.

Objects must be allocated by the standard procedure NEW before they can be used. NEW allocates memory for the fields of the object and causes the statement sequence in the class body to be executed. As with pointers, there is no explicit way to deallocate an object.

Within methods the predeclared pseudovariable SELF denotes the object for which the method has been called (the receiver of the message). SELF can also be used in class bodies to refer to the newly allocated object. It must not be used as the target of an assignment.

Although it is not necessary to qualify private fields and methods by SELF (i.e. mode and SELF.mode are equivalent within a Viewer method), it may sometimes improve readability to do so, especially for inherited fields and methods whose declarations are not at hand. A reference to SELF is only necessary when the object itself is assigned to some other object.

Example for the use of objects:

```
VAR v1, v2: Viewer;
. . .
NEW(v1); v1.Open(x0, y0, mode);
NEW(v2); v2.Open(x1, y1, mode);
. . .
v1.Close; v2.Close;
```

3.4 Declaration Order

Constant, types, variables, procedures and classes may be declared in any sequence within a block. Classes may only be declared in a module block.

```
$   DeclarationSequence    = { CONST {ConstantDeclaration ";" }
$                             | TYPE {TypeDeclaration ";"}
$                             | VAR {VariableDeclaration ";"}
$                             | ProcedureDeclaration ";" | ForwardDeclaration ";"
$                             | ClassDeclaration ";" | ClassForwardDeclaration ";"}
```

When two classes refer to each other recursively, one-pass compilation requires a forward declaration of (parts of) a class.

```
$   ClassForwardDeclaration  = CLASS "↑" ident [SuperClass] ";" DefinitionBody END ident.
$   DefinitionBody           = Fields MethodHeadings.
$   MethodHeadings           = {ProcedureHeading ";"}.
```

Fields or methods from the actual class declaration may be omitted in the forward declaration. However the fields and methods declared in *DefinitionBody* must occur as the first declarations in *Body* of the actual class declaration and must appear in the same order. The types of the fields and the parameter lists of the methods must be identical in the class declaration and in its forward declaration. If a class has a superclass its name must be specified in the class declaration as well as in the forward declaration.

3.5 Superclasses and Subclasses

A subclass inherits all fields and methods from its superclass. An inherited method can be overridden by redeclaring it in the subclass. The redeclared method must have the same parameter list as the overridden method. Overridden methods can still be accessed by the pseudovariable SUPER (see example at the end of this section). A SUPER reference in a class C denotes the object SELF with a run time type forced to the superclass of C. SUPER may only be used for qualification of methods. Fields cannot be overridden, therefore their names must be different from the names of the fields in all superclasses.

Before the body of a class is executed, the body of its superclass is executed.

The compatibility rules between superclasses and subclasses are the same as between records and extended records in Oberon. Given the two class declarations

```
CLASS C0; . . . END CO;
CLASS C1(C0); . . . END C1;
```

objects of the subclass C1 can be assigned or passed as a value parameter to objects of the superclass C0 but not the other way round. In the case of variable parameters the actual and formal parameter types must be identical. Given the following declarations

```
VAR c0: C0; c1: C1;
```

the assignment

```
c0 := c1
```

has the effect that c0 refers to the same object as c1. The static type of c0 is C0, but its dynamic type is C1. Both objects share the values of their fields. To get a physical copy of an object, one has to write the assignment

```
c0^ := c1^
```

Here, only the common fields of c0 and c1 are copied. c0 must already have been allocated. It retains the dynamic type it had before the assignment.

To summarize, the rules of assignment compatibility between objects are:

(1) An object of class C may be assigned to an object of the same class or a superclass of C
(2) NIL may be assigned to any object
(hint: this can be used to mark an object's space as free for the garbage collector.)
(3) The pseudovariables SELF and SUPER must not be the target of an assignment.

Type test, type guard and *with* statement are defined for objects and classes in an analogous way to records in Oberon.

Example for subclasses and SUPER:

```
CLASS Stack;
    s: ARRAY 128 OF INTEGER;
    sp: INTEGER;
    PROCEDURE Push (x: INTEGER); BEGIN . . . END Push;
    PROCEDURE Pop (): INTEGER; BEGIN . . . END Pop;
BEGIN sp := 0
END Stack;

CLASS CountStack (Stack);
    max: INTEGER;

    PROCEDURE Push(x: INTEGER);         (* overrides inherited Push *)
    BEGIN
        SUPER.Push(x); IF sp > max THEN max := sp END
    END Push;

BEGIN max := sp          (* body of Stack is executed prior to this body *)
END CountStack;
```

Example for the compatibility of classes:

```
VAR s0: Stack; s1: CountStack;

NEW(s0); NEW(s1);
s0 := s1; . . .
IF s0 IS CountStack THEN s1 := s0 (CountStack) END
```

3.6 Export of Classes

Classes declared in the definition part of a module are exported.

```
$   ClassDefinition          = CLASS ident [SuperClass] ";" DefinitionBody END ident.
```

Fields or methods declared in the implementation part may be omitted in the definition part. Only the names declared in the definition part are visible to clients of the module.

The fields types and the parameter lists of the methods must be identical in both declarations. If a class has a superclass its name must be specified in both class declarations.

Implementation note: The fields and methods declared in *DefinitionBody* (see Chapter 3.4) should occur as the first declarations in the corresponding implementation *Body* and should appear in the same order, otherwise the clients of the module have to be recompiled.

The sequence of definitions in the definition part of a module is as follows:

```
$   DeclarationSequence      = { CONST {ConstantDeclaration ";" }
$                              | TYPE {TypeDeclaration ";"}
$                              | VAR {VariableDeclaration ";"}
$                              | ProcedureDeclaration ";"                 [sic: ProcedureHeading]
$                              | ClassDefinition ";" | ClassForwardDeclaration ";"}
```

In definition parts there are no forward declarations of procedures, but forward declarations of classes may still be necessary, as two classes can each have fields which are objects of the other class.

3.7 Message Variables

In some situations it is necessary to send a message not only to a single object but to all objects in a data structure. The conventional solution would be to traverse the data structure and to call the appropriate method for every encountered object. This is inconvenient, however, since the traversion algorithm has to be repeated again and again. Besides, the data structure may be hidden and therefore not available for traversing. A more elegant solution is to pass the message as a parameter to a general traversion procedure which sends it to every object in the data structure.

We define a new standard type MESSAGE. Variables of this type may assume *message constants* as their values. A message constant is a message name qualified by the name of the class of which the receiving objects are supposed to be. It also has an actual parameter list. It is called a *constant* since the message is not sent immediately but stored as a whole in the message variable to be sent later on. Syntactically, a message constant is a factor:

```
$   factor                   = . . . |MessageConst.
$   MessageConst             = ClassName "." MethodName [ActualParameters].
$   ClassName                = qualident.
$   MethodName               = ident.
```

MethodName must not denote a function. *ClassName* denotes the class an object must be derived from in order to receive this message (It does not explicitly select a method from that particular class.) Message constants may only be used as value parameters or assigned to local variables. If a message variable *a* is assigned to another message variable *b* then *a* and *b* must be declared in the same scope.

In order to send a message to an object there are two standard procedures:

SEND(x,m)	*x* must designate an object and *m* must be a variable of type MESSAGE. The message *m* with the actual parameters specified in the corresponding message constant is sent to the object *x*. The class of *x* must be equal to (or a subclass of) the class specified by *ClassName* in the message constant. Messages must not be sent to SUPER.
ACCEPTS(x,m)	*x* must designate an object and *m* must be a variable of type MESSAGE. The function ACCEPTS returns TRUE if *x* "understands" the message *m*, i.e. if the class of *x* is equal to (or a subclass of) the class specified by *ClassName* in the message constant corresponding to *m*.

Example

In the Oberon system, messages can be used to implement a broadcast to all viewers on the screen. To show the deletion of a piece of text in all viewers containing this text, one could write:

```
Views.Broadcast(TFrames.Delete(beg, end), text)
```

The procedure Broadcast would then be implemented as follows:

```
PROCEDURE Broadcast(msg: MESSAGE; text: Txt.Text);
    VAR v: Viewer; frame: Frames.Frame;
BEGIN
    v := firstViewer;
    WHILE v # NIL DO    (* for all viewers on the screen *)
        frame := v.down;
        WHILE frame # NIL DO    (* for all subframes *)
            IF ACCEPTS(frame, msg) & (frame.text = text) THEN
                SEND(frame, msg)
            END;
            frame := frame.next
        END;
        v := v.next;
    END;
END Broadcast;
```

3.8 Features Available in Other Object-Oriented Languages

Object Oberon contains only a minimal set of features for object-oriented programming. Other languages such as Smalltalk [3] or C++ [4] introduce more concepts, although some of them can also be accomplished in Object Oberon.

Static binding versus dynamic binding

C++ distinguishes between static and dynamic binding of messages and methods. In Object Oberon all methods are dynamically bound, i.e., they are taken from the class an object belongs to at run time (which can be a subclass of the class with which the object has been declared). This is in contrast to static binding, where the methods are always taken from the class with which an object has been declared. Statically bound methods cannot be overridden in subclasses. In general, a call of a statically bound method is more efficient than a call of a dynamically bound method.

In Object Oberon static binding can be achieved by using normal procedures declared outside of a class instead of methods. Those procedures must receive the object to which they refer as a parameter in order to be able to access the fields of the object (explicit “self”).

Class variables and class methods

Smalltalk introduces the concept of class variables and class methods. Such items belong to a class rather than to an object of this class. Class variables can be used to store values needed by all objects of a class. Class methods are typically used to create and initialize objects. For that reason, class methods cannot belong to an object since they have to be called before the object exists.

In Object Oberon a class variable can be implemented as a global variable of the module containing the class, and a class method as a global procedure. If one considers not the class but the module as the actual fence to clients, information hiding is preserved.

4. Implementation

This section sketches the implementation decisions, mainly the run time representation of objects and classes and the code generated for operations with objects.

Object Oberon has been implemented in Oberon on a Ceres workstation [5]. We used the original Oberon compiler written by N. Wirth as our starting point. Object Oberon is a superset of Oberon. Every Oberon program can also be compiled by the Object Oberon compiler.

4.1 Run Time Representation of Objects and Classes

Objects are implemented as pointers to records containing the instance variables, a type tag and a pointer to a method table. For every class there is one such method table to which all objects of this class refer (see Fig. 1).

The type descriptor is already present in the Oberon implementation of records and can be borrowed for objects. It is used by the garbage collector and for dynamic type tests. We did not touch the information in type descriptors. For classes we added the method table, which is implemented as an array of procedure variables. It is indexed with the method number. Every entry holds a procedure descriptor (module pointer and relative program counter).

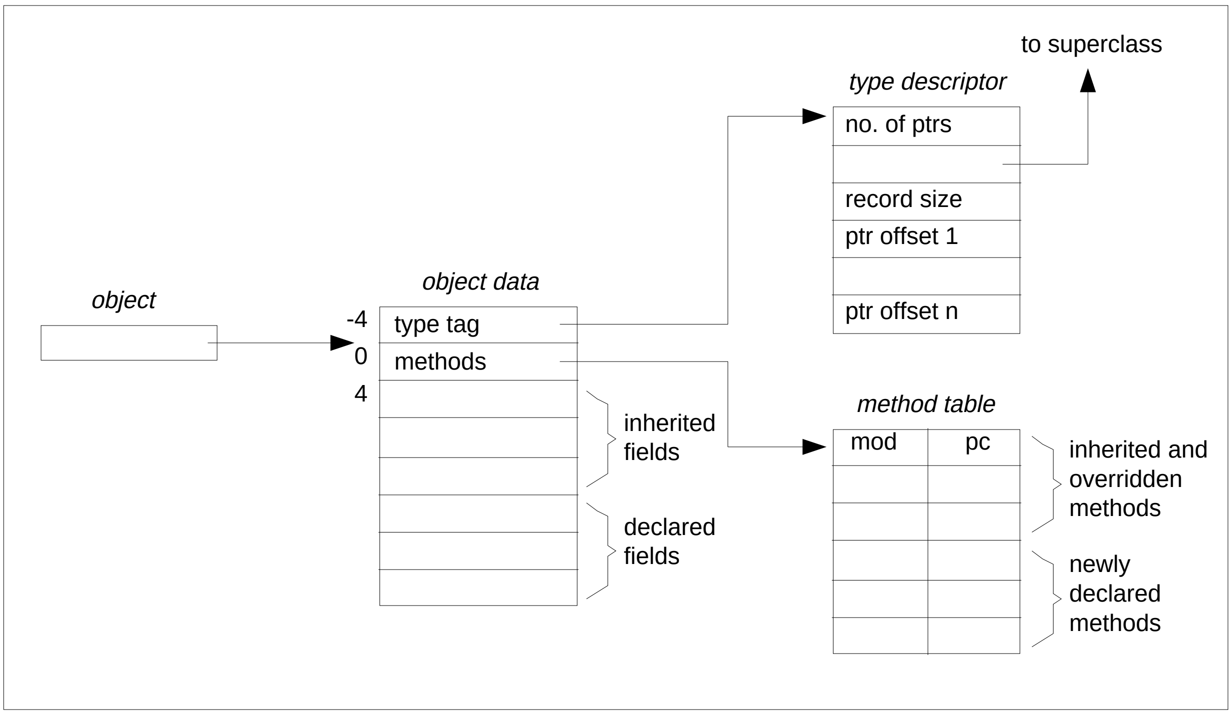


Fig. 1 Run time representation of objects and classes in Object Oberon

An alternative to a separate method table would be to store the method table with the type descriptor. This would save an extra pointer to the method table in every object. A natural solution would be to access the methods with negative offsets relative to the type descriptor. But this would have caused major changes in the allocation and handling of the descriptors. So we considered appending the method table at the end of the type descriptor. But unfortunately there is a catch, too: The offsets of inherited methods to the start of the type descriptor must be the same in the superclass and in the subclass. But this cannot be guaranteed because new pointers may be declared in the subclass causing its type descriptor to grow. A solution would be to get the length of the type descriptor from the number of pointers stored in it, and to use this as an offset whenever a method is called (this offset is not known at compile time!). This would result in the following code for a method call:

```
. . .                                   R7 = object
MOVD        -4(R7), R7                  R7 := address of type descriptor
MOVXBD      0(R7), R6                   R6 := no. of methods
CXPD        8+m(R7)[R6:W]               call method m
```

By using a separate method table we can avoid the MOVXBD instruction and the index register in CXPD, saving 4 bytes and 2 memory accesses for every method call. Another advantage is that we do not have to rely on the format of the type descriptor and are thus unaffected by future changes. The cost for this is an overhead of 4 bytes in every object. However, the same amount is also needed if a handler is installed like in the objects of the Oberon system. Besides, we believe that objects should not be used for small entities but only for sufficiently complex data with sensible operations on it. For such objects, the small storage overhead should be negligible.

4.2 Allocation of the Method Table

The method table of a class is set up during the initialization of the module containing the class declaration. It is created by the following steps:

(1) Allocate the method table on the heap;
(2) Copy descriptors of inherited methods from the method table of the superclass;
(3) Load new method entries from the link table of the processor (see [2]).

This accounts for 10 to 35 bytes (depending on the superclass) plus 5 bytes for every declared method. A pointer to the method table is stored in the constant area just behind the address of the type descriptor of that class. Whenever an object is created, this pointer has to be installed into it.

4.3 Creating an Object

When an object is created by the standard procedure NEW, the following actions are executed:

(1) Allocate object space on the heap;
(2) Install the pointer to the method table in the object;
(3) Call the body of the object's class.

This accounts for 19 bytes.

4.4 Calling a Method

Methods are like procedure variables. The method table is indexed with the method number and the procedure descriptor found there is taken as the argument of the call instruction.

the receiver of a message is implicitly passed as the first parameter of a method call.

```
MOVD      object(FP), R7                                         3
MOVD      R7, TOS           Push receiver                        2
MOVD      0(R7), R7         R7 := address of method table        3
Push parameters
CXPD      m(R7)             Call method m                        3
```

The numbers to the right give the length of the instructions: 11 bytes are required for a method call (without parameter passing). This is 5 bytes less than for the call of a handler in Oberon:

```
MOVQW     0, id(FP)             msg.id := 0                          3
Set up parameters in msg
MOVD      object(FP), TOS       Push receiver                        3
MOVD      Descriptor, TOS       Push type information about msg      3
ADDR      msg(FP), TOS          Push address of msg                  3
CXPD      handle(object(FP))    Call object^.handle                  4
```

For a method call using the pseudovariable SUPER, even shorter code can be generated because the location of the method table is known at compile time and does not have to be extracted from the object.

```
MOVD      object(FP), TOS     Push receiver                                  3
MOVD      tabAdr(SB), R7      R7 := method table address of the superclass   3
Push parameters
CXPD      m(R7)               Call method m                                  3
```

4.5 Message Variables and Message Constants

when a message constant occurs, its actual parameters are pushed on the procedure stack and a message descriptor of the following form is built.

Address of actual parameters	4
Address of class descriptor	4
Size of actual parameters	4
Method number	4

Every message variable is represented by such a descriptor. When a message stored in a message variable is sent to an object, the parameters are copied to the top of the procedure stack and the method with the stored number is called. The class descriptor is used for a type test in order to determine if the receiver object understands the message.

5. Measurements

Since the Object Oberon compiler was derived from the Oberon compiler, it makes sense to compare the sizes of the two compilers in order to determine the cost of the extension. The changes mainly affect the parser, the table handler (OCT), and parts of the code generator (OCC and OCH). The source code increased from 3956 to 4429 lines (11.9%), the object code from 42184 to 47504 bytes (12.6%). Fig 2 shows the costs of the extension in the several modules of the compiler.

It is also an interesting to look into the costs of using classes, i.e. what is the time overhead of a method call compared with other procedure calls. To make procedure calls comparable to method calls, wee passed a pointer as a parameter to procedures in the same way as it is done for methods. This is realistic, since procedures working on abstract data usually get a pointer to the data as a parameter (e.g. the procedures working on texts in the Oberon system). We measured a call to a local procedure

```
PROCEDURE P0 (obj: Ptr); BEGIN END P0;
```

a call to an external procedure

```
PROCEDURE *P1 (obj: Ptr); BEGIN END P1;
```

a call to a handler (The following is an optimistic assumption for a handler. A typical handler is much more costly: it contains type tests, nested if statements, etc.)

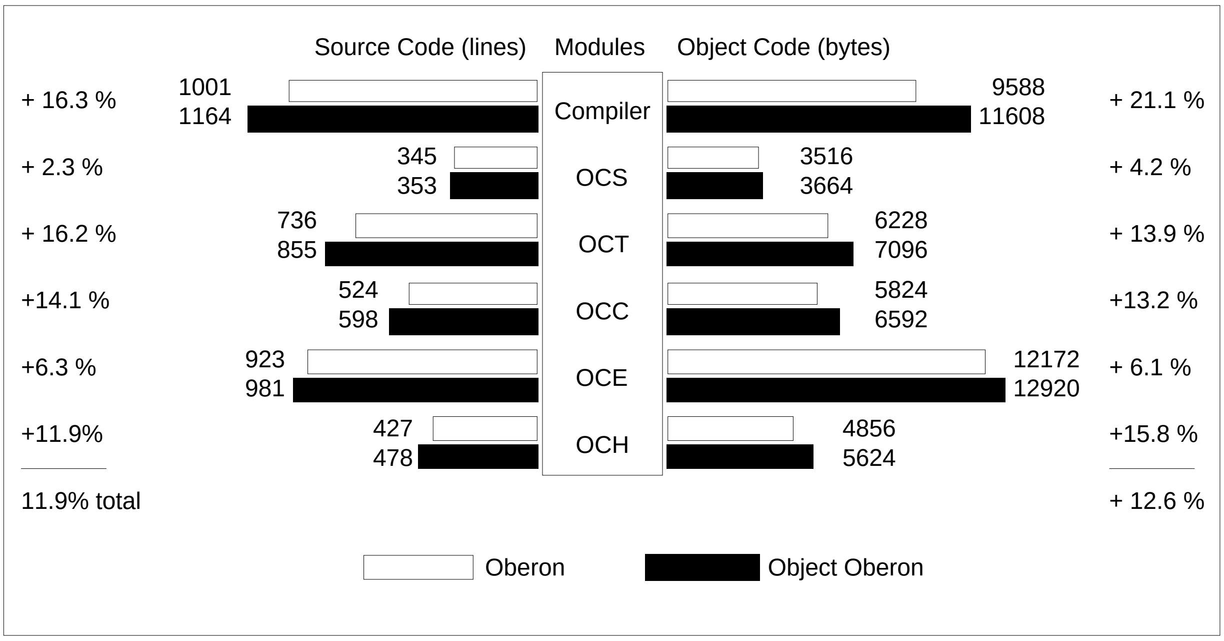


Fig. 2 Size of the Oberon compiler and the Object Oberon compiler

```
PROCEDURE *Handle (obj: Ptr; VAR msg: Message);
BEGIN
    CASE msg.id OF 0: END
END Handle;
```

and a call to a method (the receiving object is passed as an implicit parameter)

```
CLASS C;
    PROCEDURE P; BEGIN END P;
END C;
```

The declarations for the above procedures are

```
TYPE
    Message = RECORD id: INTEGER END;
    Ptr = POINTER TO Rec;
    Rec = RECORD
        handle: PROCEDURE (Ptr, VAR Message)
    END;
```

We called every procedure a million times and extracted the time for a single call. The following table shows the results:

Local procedure call	2.29 µs
External procedure call	3.69 µs
Handler call	6.13 µs
Method call	3.96 µs

The main result is that a method call is 7.5% slower than a call to an external procedure, whereas a call to a handler is 66% slower. This suggests that when objects are used extensively, programming with classes and methods is clearly more efficient than with records and installed handlers.

6. Discussion

A class is something between a module and a record. It is similar to a module because it encapsulates data and operations. It is also similar to a record because it contains structured data and can be used as a data type for variables. Classes in Object Oberon reflect both similarities. Syntactically they resemble modules, semantically they are similar to records (e.g. scope rules).

Classes and records have much in common. In fact, one could consider sacrificing records (at least extensible ones) and viewing them as special cases of classes without methods. One may argue that objects are always allocated on the heap, keeping the garbage collector busy, whereas records can be allocated on the procedure stack. But in the Oberon system, records that play a central role in a data structure are allocated on the heap anyway, and records that are better placed on the procedure stack do not appear frequently (An exception to this are message records in the current Oberon system. However, when classes and methods are available, message records are not necessary any more).

For various reasons, programming with classes is more convenient than programming with records and handlers like in the Oberon system:

- The notation for method calls is concise. No message record has to be initialized prior to the call, and the target object does not have to be passed to the method explicitly.
- The method lookup, i.e. the initialization of the method table, does not need to be programmed manually. This makes programs shorter and safer.
- Objects are initialized automatically upon creation. This serves both convenience and safety.
- From the class declaration it is obvious what operations are defined for a class. This is not so clear when a handler is used, since the correspondence between message numbers and methods is hidden in the implementation part of a module. It is easy to work with the wrong message number accidentally, especially after new message numbers have been inserted.

On the other hand using handlers instead of methods had the following advantages.

- No language extension is needed.
- If new messages are introduced, the interface to the clients is not affected (although this may be also regarded as a disadvantage!).
- Messages can be sent to a handler which do not correspond to any method. This may be an error or a desired effect. The handler can pass the message to some other handler or simply do nothing (but this may cause errors to go undetected).
- It is easy to implement a broadcast of any message to any object.

Handlers and message records are low-level constructs for object-oriented programming. They are more flexible for the same reason that the low-level facilities in Modula-2 are more flexible than the regular language constructs. Using handlers, the computer cannot check, whether a message is allowed to be sent to an object and if all required parameters have been provided. Thus, errors may result at run time which could be detected at compile time, if classes were used.

The introduction of classes has not increased the size of the compiler dramatically (12%). The storage overhead in every object is the same as the overhead for a handler in an Oberon record (4 bytes. A method call is only 7% slower than a normal procedure call, but is clearly more efficient than a call to a handler both in code size (45%) and in run time (55%).

Object Oberon is an attempt to get the maximum out of a minimal set of new constructs. This is in the spirit of Oberon and in contrast to many other object-oriented languages like Smalltalk or C++. The new constructs are classes and message variables. While classes are essential to object-oriented programming, message variables are not. They should be viewed as an experiment in providing an efficient and flexible message broadcasting mechanism, which is not available in most object-oriented languages except Smalltalk.

References

[1] Wirth N.: The Programming Language Oberon. *Software-Practice and Experience,* 18 (1988)

[2] Wirth N., Gutknecht J.: *The Oberon System.* Report 88, ETH Zürich, 1988

[3] Goldberg A., Robson D.: *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1983.

[4] Stroustrup B.: *The C++ Programming Language.* Addison-Wesley, 1986

[5] Eberle H.: *Development and Analysis of a Workstation Computer.* Ph.D. thesis, ETH Zürich, 1987

Appendix 1: Object Oberon Grammar

Extensions are printed in boldface.

Oberon	=	MODULE ident “;” [ImportList] DeclSeq [BEGIN StatSeq] END ident “.”
	|	DEFINITION ident “;” [ImportList] DefSeq END ident “.”.
ImportList	=	IMPORT ident [“:” ident] {“, ident [“:” ident]} “;”.
DefSeq	=	{ CONST {ConstDecl} | TYPE {TypeDecl} | VAR {VarDecl} | ProcHeading | **ClassDef** }.
DeclSeq	=	{ CONST {ConstDecl} | TYPE {TypeDecl} | VAR {VarDecl} | ProcDecl | **ClassDecl** }.
ConstDecl	=	ident “=” ConstExpr “;”.
TypeDecl	=	ident “=” Type “;”.
VarDecl	=	IdList “:” Type “;”.
ProcHeading	=	PROCEDURE ident [FormPars] “;”.
ProcDecl	=	PROCEDURE “^” ident [FormPars] “;”
	|	PROCEDURE [“*”] ident [FormPars] “;” DeclSeq [BEGIN StatSeq] END ident “;”.
ClassDef	**=**	**CLASS [“^”] ident [“(” Qualident “)”] “;”**
		FieldListSeq {ProcHeading} END ident “;”.
ClassDecl	**=**	**CLASS [“^”] ident [“(” Qualident “)”] “;”**
		FieldListSeq {ProcDecl} [BEGIN StatSeq] END ident “;”.
FormPars	=	“(“ [FPSection {“;” FPSection}] “)” [“:” Qualident].
FPSection	=	[VAR] IdList “:” FormalType.
FormalType	=	{ARRAY OF} Qualident.
Type	=	Qualident
	|	ARRAY ConstExpr { “,” ConstExpr} OF Type
	|	RECORD [“(“ Qualident “)”] FieldListSeq END
	|	POINTER TO Type
	|	PROCEDURE [“(“ [[VAR] FormalType {“,” [VAR FormalType}]”)” [“:” Qualident]].
FieldListSeq	=	FieldList {“;” FieldList}.
FieldList	=	[IdList “:” Type].
IdList	=	ident {“,” ident}.
ConstExpr	=	Expr.
Expr	=	SimExpr [(“=” | “#” | “<“ | “<= | “>” | “>=” | IN | IS) SimExpr].
SimExpr	=	[“+”|”-”] Term {(“+”|”-”|OR) Term}.
Term	=	Factor {(“*”|”/”|DIV|MOD}|”&”) Factor}.
Factor	=	number | string | NIL | Set | Designator [ActPars] | “(“ Expr “)” | “~” Factor.
Set	=	“{“ [Elem {“,” Elem}] “}”.
Elem	=	Expr [“..” Expr].
ActPars	=	“(“ [Expr {“,” Expr}] “)”.
StatSeq	=	Stat {“;” Stat}.
Stat	= [	Designator (“:=” Expr | [ActPars])
	|	IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq} [ELSE StatSeq] END
	|	CASE Expr OF Case {“|” Case} [ELSE StatSeq] END
	|	WHILE Expr DO StatSeq END
	|	REAPEAT StatSeq UNTIL Expr
	|	LOOP StatSeq END
	|	RETURN [Expr]
	|	EXIT
	|	WITH Qualident “:” Qualident DO StatSeq END
	].	
Case	=	[CaseLabels { “,” CaseLabels} “:” StatSeq].
CaseLabels	=	ConstExpr [“..” ConstExpr].
Designator	=	ident {“.” ident | “[“ Expr {“,” Expr} “]” | “^” | “(“ Qualident “)”}.
Qualident	=	ident [“.” ident].

Appendix 2: New Compiler Error Messages

150	Identifier does not match class name
151	classes must be declared at module level
152	base type is not a class
153	unresolved forward class definition
154	wrong order of methods between class declaration and forward declaration
155	multiple class forward definitions
156	message constant must not denote a function

Appendix 3: A Basic Class Library for Object Oberon

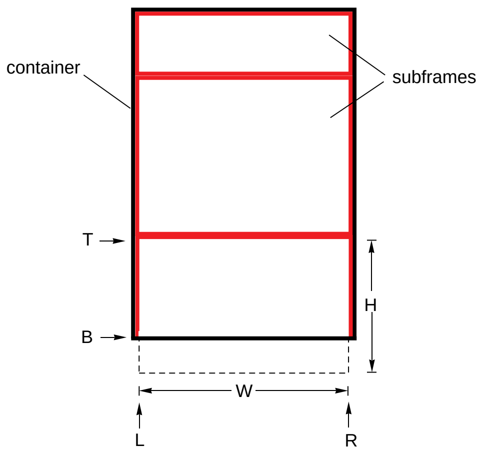
This is a basic class library to be used in Object Oberon programs. It supports viewers, various kinds of frames, text and graphics. The following modules rely on the original Oberon system modules like Display, Viewers and Texts. This is mainly to retain compatibility with the existing Oberon system. By this example we hope to demonstrate the usefulness of the class construct in building extendible software units.

Frames

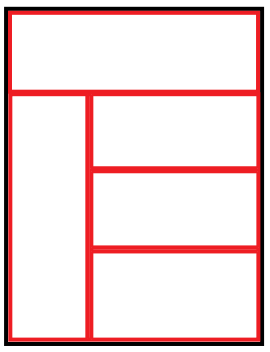
Frames are a very heavily used class. Every rectangular area on the screen is derived from a frame: viewers, menus, title bars, and even the screen itself is a frame. A frame is container for text, graphics or other frames.

```
DEFINITION Frames;
    IMPORT Objects, Files;
    CONST
        vertical = 0; horizontal = 1; relative = 2;   (* subframe allocation strategies for SetLT *)

    CLASS Frame (Objects.Object);
        L, T, R, B: INTEGER;            (* absolute frame coordinates: left, top, right, bottom *)
        W, H: INTEGER;                  (* desired width and height; default: as large as possible *)
        up, down, next: Frame;          (*container frame, first subframe, next frame *)
        visible: BOOLEAN;               (* TRUE if (partially) visible *)
        PROCEDURE Install(f: Frame);
        PROCEDURE Remove(f: Frame);
        PROCEDURE SetLT(f: Frame; strategy: SHORTINT);

        PROCEDURE Show;
        PROCEDURE Hide;
        PROCEDURE Resize(r, b: INTEGER);
        PROCEDURE Move(dx, dy: INTEGER);
        PROCEDURE Copy(VAR f: Frame);

        PROCEDURE Defocus;
        PROCEDURE Neutralize;
        PROCEDURE HandleKey(ch: CHAR);
        PROCEDURE HandleMouse(x,y: INTEGER; buttons: SET);

        PROCEDURE This(x, y: INTEGER): Frame;
        PROCEDURE Broadcast(m: MESSAGE);
        PROCEDURE Load(VAR r: Files.Rider);
        PROCEDURE Store(VAR r: Files.Rider);
    END Frame;

    VAR
        Screen: Frame;                  (* the whole screen *)
        Focus: Frame;                   (* current focus frame *)

    PROCEDURE SetFocus(f: Frame);
    PROCEDURE RemoveMarks(f: Frame);
END Frames.
```

Every frame has a physical and logical size. The physical size (R-L, B-T) is the size of the frame as it is displayed on the screen. The logical size (W, H) specifies the maximum size a frame can take if the container frame is large enough. It may extend the physical size to the right and the bottom (Fig a). A more complicated example for nested frames is shown in Fig b.



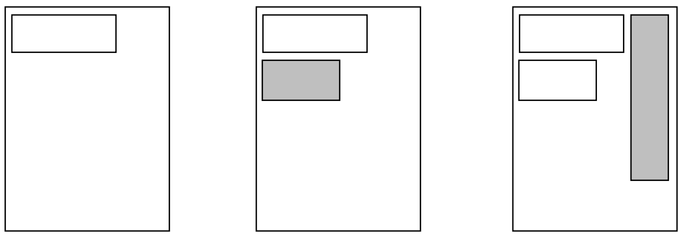
(a)


(b)

One of the frames is the focus frame which receives the characters typed in from the keyboard. It must be a subframe of the current focus viewer. It is the programmer's responsibility to set the focus frame accordingly.

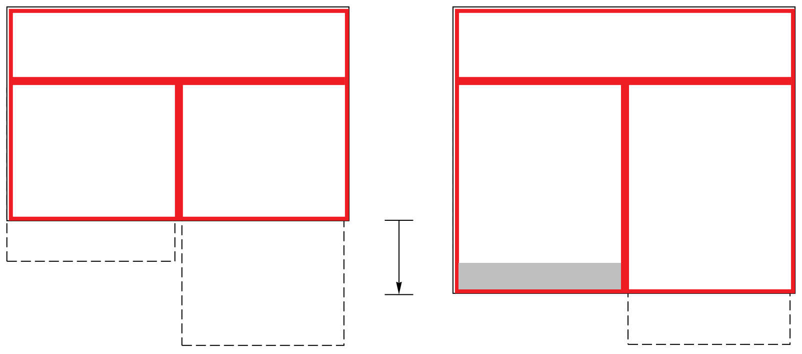
Most of the following methods are called automatically by the system (e.g. Show is called when a frame is installed). Their main purpose is to keep the frame data consistent and to propagate messages to the subframes. In subclasses, the *Frames* methods usually have to be overridden (extended).

• *Install(f)* installs the frame *f* as a subframe into the receiver according to the values of *L*, *T*, *W*, and *H*. The values of *R* and *B* are calculated. The installed frame gets a *Show* message.
• *Remove(f)* removes the subframe *f* from the receiver. The container frame is not restored and the other subframes are not affected.
• *SetLT(f, s)* calculates the top left corner of the frame *f* according to the strategy *s*.
• *s* = vertical: *f* is placed below the previously installed subframes.
• *s* = horizontal: *f* is placed to the right of the previously installed subframes.
• *s* = relative: before the call (*L*, *T*) of f are given relative to (*L*, *T*) of the container. The result is (*L*, *T*) of *f* in absolute coordinates.


s = vertical

s = horizontal

- *Show* displays the data of the frame and is forwarded to all subframes. (In the *Frames* class there is nothing to be displayed, but subclasses may override *Show* to display their data.) *Show* is automatically called when a previously hidden frame becomes visible.
- *Hide* is called when the frame becomes invisible (closed or overlayed). It is forwarded to all subframes. Subclasses may override it to release data structures that are not needed when the frame is invisible.
- *Resize(r, b)* changes the right bottom corner of the frame to *r* and *b* but does not restore the frame contents on the screen. The message is forwarded to all partially visible subframes. The following figure shows the effect of resizing the bottom of a frame. (Dotted lines denote the logical size of subframes. A frame never exceeds its logical size.)



If a subframe becomes hidden by a *Resize* operation, it gets the message *Hide*. If a hidden subframe becomes visible, it gets the message *Show*.

- *Move(dx, dy)* moves the frame and its subframes by *dx* and *dy* without updating the screen.
- *Copy(f)* returns a deep copy of the receiver in *f*. If called with *f*= NIL the copy gets the class of the receiver.
- *Defocus* sets the current focus frame to NIL.
- *Neutralize* removes all user marks in the frame and its subframes (empty in this class).
- *HandleKey(ch)* is sent to the current focus frame whenever a character *ch* has been entered from the keyboard.
- *HandleMouse(x,y, buttons)* is called repeatedly while the mouse is inside the frame. If there is a subframe containing *x*, *y* the message is forwarded to this subframe.
- *This(x, y)* returns the subframe containing the point (*x, y*) or NIL if there is no such subframe.
- *Broadcast (m)* sends the message *m* to all visible subframes.
- *Load(r)* sends Load messages to all subframes to read their contents from the rider *r*.
- *Store(r)* sends Store messages to all subframes to write their contents to the rider *r*.

- *SetFocus(f)* makes *f* the new focus frame. The old focus is defocused.
- *RemoveMarks(f)* removes all system marks (cursor and star-shaped marker) from *f*.

The screen is also a frame that contains all visible viewers and its subframes. It can be used, for example, for the following operations:

Screen.This(x,y) returns the viewer containing (*x,y*)
Screen.Show sends Show messages to all viewers
Screen.Neutralize removes the user marks in all viewers

Viewers

A viewer is a frame with a border line. It normally contains a title bar and a menu (also frames). It can be resized, copied, expanded to screen height or width, stored and closed. Normally, a viewer should not be used as a drawing area, but it should contain frames in which text or graphics is drawn. The same kind of viewers can be used for all kinds of data displayed on the screen, so a text viewer is no different from a graphic viewer, it only contains a different kind of frame.

```
DEFINTION Views;
    IMPORT Frames;

    CONST userTrack = 0; systemTrack = 1;

    CLASS Viewer(Frames.Frame);
        (* overridden methods *)
        PROCEDURE Show;
        PROCEDURE Resize(r, b: INTEGER);
        PROCEDURE Move(dx, dy: INTEGER);
        PROCEDURE Copy(VAR f: Frames.Frame);
        PROCEDURE Install(f: Frames.Frame);
        PROCEDURE Remove(f: Frames.Frame);
        PROCEDURE SetLT(f: Frames.Frame; strategy: SHORTINT);
    END Viewer;

    PROCEDURE New(title, menu: ARRAY OF CHAR; strategy: SHORTINT): Viewer;
    PROCEDURE Title(title: ARRAY OF CHAR): Frames.Frame;
    PROCEDURE Menu(manu: ARRAY OF CHAR): Frames.Frame;

    (* standard commands of a viewer menu *)
    PROCEDURE Close;
    PROCEDURE Copy;
    PROCEDURE Grow;
    PROCEDURE Store;
END Views.
```

- *Show* draws the border and the title bar.
- *Resize(r, b)* restores the border of the viewer. Any extended part of the viewer is erased.
- *Move(dx, dy)* moves the viewer bitmap by *dx* and *dy*.
- *Copy(f)* returns a deep copy of the viewer. If *f* = NIL before the call, it gets the class Viewer.
- *Install(f)*, *Remove(f)* and *SetLT(f, s)* are like in Frames.Frame but consider also the border line of the viewer.

- *New(title, menu, s)* returns and displays a new viewer in the tack specified by s. If *title* # “” a title frame with the given name is installed as the first subframe. If *menu* # “” a menu with the given menu text is installed as the second subframe. If the star-shaped marker is set, it is used to determine the top of the new viewer, otherwise a default value is chosen for the top.
- *Title(t)* returns a title frame with the text *t*.
- *Menu(m)* returns a static text frame with the text *m*.
- *Close* closes the viewer containing this command. If it is the only viewer in its track and the track is an overlay, the track is closed and the overlayed viewers are revealed.
- *Copy* copies of the viewer containing this command
- *Grow* expands the viewer to full screen height or, if it has already this height, to full screen width.
- *Store* stores the contents of the viewer under the file name contained in the title bar and creates a backup *.Bak.

Text Frames

A text frame displays text with various fonts on the screen. Text frames already support most of the standard behaviour of a text editor (e.g. typing, selecting, copying, deleting, scrolling, etc.).

```
DEFINITION TFrames;
    IMPORT Frames, Txt, Fonts, Files

    CONST replaced = 0; inserted = 1; deleted = 2; (* update operations *)

    TYPE
        Location = RECORD
            org, pos: LONGINT;          (* org: line origin; pos: character position *)
            dx, x, y: INTEGER;          (* screen position (relative to frame); dx: width of character at x, y *)
        END;

    CLASS Frame (Frames.Frame);
        text: Txt.Text;                 (* displayed Text *)
        org: LONGINT;                   (* position of first displayed character *)
        lsp, asr, dsr: INTEGER;         (* line space, ascender, descender of displayed text *)
        margW: INTEGER;                 (* left margin (scroll bar) *)
        time: LONGINT;                  (* time of last selection *)
        caret: Location;                (* caret location; caret.pos < 0 if caret not set *)
        selbeg, selend: Location;       (* selection range; selbeg.pos < 0 if no selection set *)
        autoInd: BOOLEAN;               (* auto indent for Tabs (default: TRUE) *)

        PROCEDURE Update(beg, end: LONGINT; mode: INTEGER);
        PROCEDURE Delete(beg, end: LONGINT);
        PROCEDURE Insert(pos: LONGINT);
        PROCEDURE Write(ch: CHAR);
        PROCEDURE SetFont(beg, end: LONGINT; fnt: Fonts.Font);
        PROCEDURE SetColor(beg, end: LONGINT; color: SHORTINT);
        PROCEDURE SetCaret(pos: LONGINT);
        PROCEDURE RemoveCaret;
        PROCEDURE SetSelection(begin, end: LONGINT);
        PROCEDURE RemoveSelection;

        PROCEDURE TrackCaret(VAR x, y: INTEGER; VAR buttons: SET);
        PROCEDURE TrackSelection(VAR x, y: INTEGER; VAR buttons: SET);
        PROCEDURE TrackLine(VAR x, y: INTEGER; VAR org: LONGINT; VAR buttons: SET);
        PROCEDURE TrackWord(VAR x,y: INTEGER; VAR pos: LONGINT; VAR buttons: SET);

        PROCEDURE ShowFrom(pos: LONGINT);
        PROCEDURE Pos(x, y: INTEGER): LONGINT;
        PROCEDURE MarkBusy;

        (* overridden methods *)
        PROCEDURE Defocus;
        PROCEDURE Neutralize;
        PROCEDURE Show;
        PROCEDURE Hide;
        PROCEDURE Resize(r, b: INTEGER);
        PROCEDURE Copy(VAR f: Frames.Frame);
        PROCEDURE HandleMouse(x, y: INTEGER; buttons: SET);
        PROCEDURE HandleKey(ch: CHAR);
        PROCEDURE Load(VAR r: Files.Rider);
        PROCEDURE Store(VAR r: Files.Rider);
    END Frame;
END TFrames.
```

A newly allocated text frame is initialized to its default values but no text is installed. When a subclass is derived from *TFrames* one usually has to override the two methods *HandleMouse* and *HandleKey* to handle user Input in a special way. In most cases the rest can be inherited as it is.

- *Update(beg, end, m)* restores the screen after the range *beg..end* has been modified in the text according to the operation *m.* Called from *Insert*, *Delete*, *Write*, *SetFont*, and *SetColor*.
- *Delete(beg, end)* deletes the text range beg..end and restores the screen.
- *Insert(p, b)* inserts the buffer *b* at the position *p* in the text and restores the screen.
- *Write(ch)* inserts the character *ch* at the caret position into the text and shows it on the screen in the font of the preceding character.
- *SetFont(beg, end, f)* changes the font of the range *beg..end* to *f* and restores the screen.
- *SetColor(beg, end, c)* changes the color of the range *beg..end* to *c* and restores the screen.
- *SetCaret(pos)* sets the caret to position *pos* and makes the receiver the current focus frame. It also sets the Oberon focus viewer. A previously displayed caret is removed.
- *RemoveCaret* removes the caret if it is set.
- *SetSelection(beg, end)* selects the range *beg..end.* An old selection in this frame is removed.
- *RemoveSelection* removes the selection if it exists.
- *TrackCaret(x, y, b)* sets the caret at screen position *x, y* and tracks it until all mouse buttons are released. The buttons pressed during tracking are returned in *b* (left=2, middle=1, right=0).
- *TrackSelection(x, y, b)* The right mouse button has been pressed at *x, y*. Text is selected according to the mouse movement until all buttons are released. The buttons pressed during tracking are returned in *b.*
- *TrackLine(x, y, org, b)*. The mouse is at *x, y.* Mouse movements are tracked and the line containing the current *y* is underlined until all buttons are released. On completion *org* is the position of the first character in this line and *b* is the set of buttons pressed during tracking.
- *TrackWord(x, y, pos, b)*. The mouse is at *x, y.* Mouse movements are tracked and the word containing the current *x, y* is underlined until all buttons are released. On completion *pos* is the position of the underlined word and *b* is the set of buttons pressed during tracking. A word is a string of characters greater than blank.
- *ShowFrom(pos)* redraws the text from the first line start following *pos*.
- *Pos(x, y)* returns the text position corresponding to the screen coordinates *x, y.*
- *MarkBusy* toggles an arrow at the bottom of the scroll bar on and off.

Overridden methods (see descriptions in the superclass):

- *Defocus* removes the caret from the receiver and sets the current focus frame to NIL.
- *Neutralize* removes all text marks from the receiver (caret, selection, scroll mark)
- *Show* redraws the whole text of the frame from the position org.
- *Hide* removes all marks in the receiver and releases internal data structures.
- *Resize(r, b)* extends the right and bottom margin of the receiver to *r* and *b* respectively and redraws the text in any extended part of the frame.
- *Copy(f)* returns a deep copy of the receiver in *f.* If *f*= NIL before the call, it gets the class of the receiver.
- *HandleMouse(x, y, b)* draws the mouse pointer while the mouse is inside the frame. On a mouse click (*b* # {}) it reacts appropriately by scrolling, setting the caret, or selecting. It also handles the shortcuts for deleting and copying selected text.
- *HandleKey(ch)* is called when a character *ch* has been typed while the receiver is the focus frame. It writes *ch* at the caret position.
- *Load(r)* reads a text from the rider *r* and installs it into the frame without displaying it.
- *Store(r)* writes the text in the frame to the rider *r.*

Static Text Frames

The following module provides a class for static text frames, i.e. frames that display text but do not support scrolling and edition. Menu frames or title bars of viewers are examples of such frames.

```
DEFINITION STFrames;

    IMPORT TFrames, Files;

    CLASS Frame (TFrames.Frame);
        (* overridden methods *)
        PROCEDURE HandleMouse(x, y: INTEGER; buttons: SET);
        PROCEDURE HandleKey(ch: CHAR);
        PROCEDURE Copy(VAR f: Frames.Frame);
        PROCEDURE Load(VAR r: Files.Rider);
        PROCEDURE Store(VAR r: Files.Rider);
    END Frame;

END STFrames.
```

Texts

In this class library module *Txt* merely establishes an object-oriented interface to the module *Texts* of the original Oberon system. It was not reasonable to reimplement Texts since existing applications like the compiler rely on the original text module and would not run with modified texts. The semantics of the classes in *Txt* is the same as in the original Oberon system and is therefore not described further.

```
DEFINITION Txt;

    IMPORT Objects, Files, Fonts, Texts;

    CONST Name = 1; String = 2; Int = 3; Real = 4; LongReal = 5; Char = 6;

    CLASS Buffer (Objects.Object);
        b: Texts.Buffer;
        len: LONGINT;
        PROCEDURE Open;
        PROCEDURE CopyTo(DB: Buffer);
    END Buffer;

    Class Text (Objects.Object);
        t: Texts.Text;
        len: LONGINT;
        PROCEDURE Append(B: Buffer);
        PROCEDURE ChangeFont(beg, end: LONGINT; fnt: Fonts.Font);
        PROCEDURE ChangeColor(beg, end: LONGINT; col: SHORTINT);
        PROCEDURE ChangeOffset(beg, end: LONGINT; voff: SHORTINT);
        PROCEDURE Delete(beg, end: LONGINT);
        PROCEDURE Insert(pos: LONGINT; B: Buffer);
        PROCEDURE Load(VAR R: Files.Rider);
        PROCEDURE Open(name: ARRAY OF CHAR);
        PROCEDURE Save(beg, end: LONGINT; B: Buffer);
        PROCEDURE Store(VAR W: Files.Rider);
    END Text;
    CLASS Reader (Objects.Object);
        r: Texts.Reader;
        PROCEDURE Open(T: Text; pos: LONGINT);
        PROCEDURE Pos(): LONGINT;
        PROCEDURE Read(VAR ch: CHAR);
    END Reader;

    CLASS Writer (Objects.Object);
        w: Texts.Writer;
        PROCEDURE Reset;
        PROCEDURE Buf(): Buffer;
        PROCEDURE SetFont(fnt: Fonts.Font);
        PROCEDURE SetColor(col: SHORTINT);
        PROCEDURE SetOffset(voff: SHORTINT);
        PROCEDURE Write(ch: CHAR);
        PROCEDURE WriteLn;
        PROCEDURE WriteInt(x: LONGINT; n: LONGINT);
        PROCEDURE WriteHex(x: LONGINT);
        PROCEDURE WriteString(s: ARRAY OF CHAR);
        PROCEDURE WriteReal(x: REAL; n: INTEGER);
        PROCEDURE WriteLongReal(x: LONGREAL; n: INTEGER);
        PROCEDURE WriteLongRealHex(x: LONGREAL);
        PROCEDURE WriteRealFix(x: REAL; n, k: INTEGER);
        PROCEDURE WriteRealHex(x: REAL);
    END Writer;

    CLASS Scanner (Reader);
        nextCh: CHAR;
        line: INTEGER;
        class: INTEGER;
        i: LONGINT;
        x: REAL;
        y: LONGREAL;
        c: CHAR;
        len: SHORTINT;
        s: ARRAY 32 OF CHAR;
        PROCEDURE Open(T: Text; pos: LONGINT);
        PROCEDURE Scan;
    END Scanner;

    VAR Log: Text;          (* text in the log viewer *)


    PROCEDURE Recall(VAR B: Buffer);
    PROCEDURE ParText(): Text;        (* equivalent to Oberon.Par.text *)
    PROCEDURE ParPos(): LONGINT;(* Oberon.Par.pos *)
END Txt.
```

Graphic Frames

Graphic frames are a simple extension of standard frames supporting some graphic drawing primitives within a rectangular clipping area. In order not to deal with “low-level” programming of frame behaviour, the methods *Show* and *Resize* set an internal clipping rectangle and call the *Restore* method. One simply has to override the *Restore* method by a procedure which redraws the entire frame. Every time a part of the frame becomes invalid (e.g. by a *Resize* operation), the clipping rectangle will be set accordingly and the *Restore* method will be called.

```
DEFINITION GFrames;
    IMPORT Cursors, Frames;

    CONST
        black = 0; white = 15;              (* standard drawing colors *)
        replace = 0; paint = 1; invert = 2; (* drawing modes *)
        left = 2; middle = 1; right = 0;    (* mouse buttons *)
        none = 0;                           (* use no pattern for Fill-operations *)
        unit = 36000;                       (* one pixel *)

    TYPE
        Pattern = LONGINT;

    PROCEDURE Grey(density: REAL): Pattern;

    CLASS Frame(Frames.Frame);
        x0, y0: INTEGER;                    (* origin position, relative to frame left top; default = (0, 0) *)
        grid: LONGINT;                      (* cursor grid; defaults = unit *)
        zoom: SHORTINT;                     (* zoom power; default = 0 *)
        marker: Cursors.Marker;             (* cursor marker; default = Cursors.Arrow *)

        PROCEDURE SetOrigin(x, y: INTEGER);
        PROCEDURE SetGrid(g: LONGINT);
        PROCEDURE SetZoom(z: SHORTINT);

        PROCEDURE TrackMouse(VAR x, y: INTEGER; VAR buttons: SET);
        PROCEDURE EditFrame(x, y: INTEGER; buttons: SET);
        PROCEDURE SetZoom(z: SHORTINT);

        PROCEDURE TrackMouse(VAR x, y: INTEGER; VAR buttons: SET);
        PROCEDURE EditFrame(x, y: INTEGER; buttons: SET);
        PROCEDURE Restore(l, t, r, b: LONGINT);
        PROCEDURE Clear;

        PROCEDURE DrawDot(x, y: LONGINT; color, mode: SHORTINT);
        PROCEDURE DrawLine(x1, y1, x2, y2: LONGINT; color, mode: INTEGER);
        PROCEDURE DrawRect(l, t, r, b, w: LONGINT; color, mode: SHORTINT; pattern: Pattern);
        PROCEDURE DrawCircle(x, y, d: LONGINT; color, mode: SHORTINT);
        PROCEDURE DrawEllipse(x, y, a, b: LONGINT; color, mode: SHORTINT);
        PROCEDURE FillRect(l, t, r, b: LONGINT; color, mode: SHORTINT; pattern: Pattern);
        PROCEDURE FillCircle(x, y, d: LONGINT; color, mode: SHORTINT; pattern: Pattern);
        PROCEDURE FillEllipse(x, y, a, b: LONGINT; color, mode: SHORTINT; pattern: Pattern);
        PROCEDURE CopyRect(l, t, r, b, dx, dy: LONGINT; mode: SHORTINT);

        (* overridden methods *)
        PROCEDURE Show;
        PROCEDURE Hide;
        PROCEDURE Resize(r, b: INTEGER);
        PROCEDURE Move(dx, dy: INTEGER);
        PROCEDURE Copy(VAR f: Frames.Frame);
        PROCEDURE HandleMouse(x, y: INTEGER; buttons: SET);
    END Frame;

END GFrames.
```

All class fields are read only except *marker*, which denotes the cursor used within the graphic frame; *marker* may be changed by the user whenever he wants to.

- *SetOrigin(x, y)* sets the origin to (*x, y*) relative to the left top corner of the frame. All other coordinates are relative to this origin.
- *SetGrid(g)* sets the grid to *g*.
- *SetZoom(z)* zooms the frame According to the factor 2^z.
- *Trackmouse(x, y, buttons)* may be called by the user (e.g. in *EditFrame*) to track the mouse and to get its actual (grid aligned) coordinates (*x, y*) and the buttons pressed. The mouse grid itself is aligned to the frame origin.
- *EditFrame(x, y, buttons)* is called automatically by the system whenever the mouse is within the frame and at least one mouse button was pressed. Typically an overridden *EditFrame* method uses *Trackmouse* in order to perform interactive editing. *EditFrame* is empty in this class.
- *Restore(l, t, r, b)* is called automatically by the system. It sets the clipping frame to *l*, *t*, *r*, *b*.
- *Clear* erases the frame.

- *Show* causes the whole frame to be redrawn by calling *Restore*.
- *Hide* sets the internal clipping rectangle to zero extension.
- *Resize(r, b)* causes the extended part of the frame to be redrawn by calling Restore after setting the clipping rectangle appropriately.
- *Move(dx, dy)* moves the frame including its origin by *dx, dy* without drawing it on the screen.
- *Copy(f)* returns a deep copy of the receiver in *f*. If *f*=NIL before the call, it gets the class of the receiver.
- *HandleMouse(x, y, b)* draws the mouse cursor and calls *EditFrame* when a button is pressed.

A Note Regarding this Version

The original version of TR 109 is out of print and not available electronically. It is of historical interest in the development of the Oberon family of languages. It is now in electronic form for the benefit of the interested. Every effort has been made to stay faithful to the original, even to the point of replicating its layout and typography.

B. Smith-Mannschott
bsmithma@iiic.ethz.ch

Note, that Object Oberon was designed at a time when the Oberon language differed from what is now known as Oberon-1:

- Separate DEFINITION and MODULE files, as in Modula-2.
- Procedure types do not have a full argument list, but rather just a list of the types of their arguments.
- Implicit pointer dereferencing does not seem to be present.
- The FOR loop is not present.
- No nested comments.

1999.03.09/10:17