Hanspeter Mössenböck

Object-Oriented Programming in Oberon-2

Second Edition

Contents

Foreword

Without a doubt the idea of object-oriented programming has brought some motion into the field of programming methodology and enlarged the set of programming languages. Object-oriented programming is nothing new—it first arose in the sixties. The motivation came from the simulation of discrete event systems. The concept first manifested itself in the language Simula 67. It took nearly two decades for the method to gain impetus, and today object-oriented programming is an important concept and a powerful technique. Meanwhile, we can even speak of an over-reaction, for the concept has become a buzzword. But buzzwords always appear where there is the hope of exploiting ill-informed clients because they see the new approach as the solution to all their problems. Thus object-oriented programming is often hailed as a panacea. And so the question is justified: What is really behind it?

To let the cat out of the bag: There is more to object-oriented programming than merely putting data as objects in the foreground, instead of algorithms to which the data are subject. It is more than purely an alternative view of programmed systems. To identify the essence of object-oriented programming, is the subject of this book. This is a textbook that shows in a didactically skillful way which concepts and constructs are new, where they can be employed reasonably, and what advantages they offer. For, not all programs are automatically improved by merely recasting them in an object-oriented style. On the contrary, the new method can only be applied sensibly where complex data structures are present. It would be unwise to prematurely discard the conventional view.

It is to the author's credit that he introduces the concepts of object-oriented programming in a constructive way, demonstrates them in an evolutionary manner, and uses suitable examples to show how these concepts can be employed judiciously. The pro-

gramming language Oberon-2 provides an excellent foundation because it adds only the few typically object-oriented concepts to those of conventional procedural programming but no more. The reader should always be aware that not the language but the methodology and discipline constitute the essential concern of the book. The language only serves the purpose of formulation in a clear and concise manner. We speak of a language supporting a method; Oberon-2 supports object-oriented programming.

The object-oriented paradigm holds so much promise especially for complex systems, because the technique of object-oriented programming makes it possible to create modular systems that are truly extensible. By extensible we mean that not only new operations can be added that build on old ones, but that the same is true for data types and their instances. These comments indicate that object orientation comes to full fruition only when combined with modularity and strict typing of data.

This book is a well-organized introduction to this new field. It is obvious that the author draws on a wealth of experience gained in years of intensive work in the area and in successful teaching. The book is an enrichment for anyone interested in modern programming techniques.

Niklaus Wirth, Zurich, 1993

Preface

Object-oriented programming (OOP) has become a buzzword that is prominently displayed in numerous journals and advertisements. What is OOP all about? Is it merely a marketing fad, or does it really denote something new and useful, perhaps even a new panacea?

To be short, OOP is no panacea. Contrary to the claims made by some vendors, it does not make programming a trivial task. OOP requires a sizable portion of ability and experience—perhaps even more than traditional programming techniques do. However, OOP definitely has its strengths: it often permits more elegant solutions than are possible with conventional techniques; it promotes modularity and thus readability and maintainability of programs; and it contributes to the extensibility and reusability of software.

This book is aimed at students of computer science as well as at practitioners who want to gain a perspective on new software development methods. Since more and more languages are being extended to include object-oriented features, this book also addresses programmers who want to make better use of these new features.

The goal of this book is to convey the fundamentals of OOP, namely classes, inheritance and dynamic binding. The emphasis is on the concepts rather than on the specifics of a particular programming language. In addition, readers should learn to determine for which problems OOP is most suitable, and which problems would be better solved with conventional means.

Object-oriented programming is programming in the large. Although its principles can be explained on the basis of small examples, wider reaching examples are necessary in order to convey the power and elegance of this technique. This is precisely what is missing in most books on the subject. Chapter 11 thus

presents the design and implementation of an adequately large system, including source code, in order to drive home the ideas behind object-oriented programming.

The examples in this book are not coded in any of the widespread languages such as Smalltalk or C++. Instead, *Oberon-2*, a language in the tradition of Pascal and Modula-2, was selected. The reason for this choice is that Oberon-2 is more compact than most of the other object-oriented languages; in fact, it is even smaller than Pascal, which makes it possible to master the language quickly. Object-oriented elements are smoothly integrated into the language without displacing proven constructs such as records, arrays and procedures. Once the reader has understood the concepts presented in this book, it should be easy to transfer them to any other language.

However, if the reader takes a liking to Oberon-2, the Oberon System, complete with compiler, editor and several other tools, can be obtained at no charge. Implementations are available for several platforms (see Appendix D). The case study printed in Chapter 11 is also available as source code.

The Oberon System was developed by Professors *Niklaus Wirth* and *Jürg Gutknecht* 1985-1987 at ETH Zürich [WiG92]. It consists not only of the Oberon language, but also of an operating system with the same name. The design of Oberon reflects the experience of the man who developed Algol W, Pascal and Modula-2. In Oberon-2, the author of this book added several extensions to the Oberon language that make it more suitable for object-oriented programming.

This book is neither a general introduction to programming nor a handbook for Oberon-2; these tasks are covered by other texts [ReW92, Rei91]. The reader is assumed to be familiar with an imperative language such as Pascal or Modula-2. Chapter 2 explains Oberon-2 only enough to enable comprehension of the examples in this book. Appendix A contains the complete language definition.

I want to express both gratitude and admiration for the two designers of Oberon for their elegant design of the operating system and the language, as well as for the ergonomic and efficient implementation that makes working with Oberon a pleasure.

I owe many of the examples to my assistants, *Robert Griesemer*, *Clemens Szyperski* and *Josef Templ*. Josef Templ also contributed valuable ideas for Oberon-2.

Last but not least, I want to thank Prof. *Peter Rechenberg*, Prof. *Jörg R. Mühlbacher*, Dr. *Martin Reiser*, Dr. *Günther Blaschek*, and Dr. *Erich Gamma* for their careful reading of the manuscript and for their numerous suggested improvements.

Zürich, 1993 Hanspeter Mössenböck

1 Overview

What is the essence of object-oriented programming? What are its typical applications, and what benefits can we expect from it? How does object-oriented thinking differ from traditional, procedure-oriented thinking? These are the questions that will be explored in this chapter.

1.1 Procedure-Oriented Thinking

Decomposing programs into procedures

Since the beginnings of programming we have been used to thinking in a procedure-oriented way. We decompose programs into procedures that transform input data into output data (Fig. 1.1).

Fig. 1.1 Procedure-oriented thinking

In order to compute the area of a figure *f*, we write

```
a := Area(f)
```

The procedure *Area* is the focus of attention, while the data *a* and *f* tend to be relegated to the background.

Problems

This approach is quite practical and usually leads to good programs. However, problems arise when a program has to deal with *several* kinds of figures (e.g., rectangles, triangles, circles, etc.). In conventional languages, it is not possible to use the same procedure for different figure types; instead, a separate procedure is required for each kind of figure (e.g., *RectangleArea*, *TriangleArea*, *CircleArea*, etc.).

Furthermore, wherever a surface area is to be computed, the various kinds of figures must be differentiated and the respective procedure must be invoked. We have to write something like

```
IF f is rectangle THEN a := RectangleArea(f)
ELSIF f is triangle THEN a := TriangleArea(f)
ELSIF f is circle THEN a := CircleArea(f)
END
```

This means that we have to perform an extensive case analysis, which not only inflates the code but also causes the types of figures to be statically bound to the program. If later on ellipses are to be handled due to changed requirements, a new case will have to be inserted *at every location* where the computation of an area occurs:

```
...
ELSIF f is ellipse THEN a := EllipseArea(f)
...
```

Modifications of this nature are troublesome and easy to overlook.

Finally, the data type for figures would also need to be changed in order to accommodate ellipses. This could mean that all programs that use figures would have to be adapted to the new type or at least to be recompiled.

1.2 Object-Oriented Thinking

Decomposition into objects that fulfill contracts

The object-oriented way of thinking focuses on the data rather than on the procedures. The data and the operations applicable to them constitute *objects* that can be asked to handle certain requests and to return data as their result (Fig. 1.2).

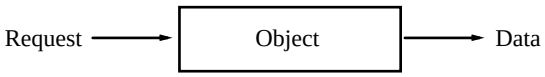

Fig. 1.2 Object-oriented thinking

The point here is that one does not have to bother about the type of the object to which a request is sent. Every type of object handles the request in its own way and carries out the correct operation: rectangles handle *Area* by computing the area of a rectangle, circles by computing the area of a circle, etc. Special notation is used to express this view. The statement

```
a := f.Area()
```

means that figure *f* is asked to handle an *Area* request. We also say that we send *f* the message *Area*. It does not matter whether *f* is a rectangle, a triangle, or a circle. Even if we later add ellipses as an additional type of object and then assume that *f* is an ellipse, the statement *a* := *f.Area*() remains unchanged. The statement is properly executed as long as ellipses understand the message *Area*. This means that the introduction of ellipses *does not affect existing code*.

Our small example already suggests some of the advantages of object-oriented programming: Object-oriented programs have to contend less with case analysis and are more extensible than procedure-oriented programs.

1.3 Object-Oriented Languages

Our next question is: What is an object-oriented programming language? This is not as easy to answer as it seems. Common OOP languages differ in many details that are not by any means all necessary for object-oriented programming. Which minimum set of features must a language provide in order to qualify as object-oriented? The most significant features are information hiding, data abstraction, inheritance, and dynamic binding.

Information hiding

Information hiding means that the implementation of complex data is encapsulated in objects and that clients have access only to an abstract view of it (Fig. 1.3). Clients cannot directly access the encapsulated data, but must use procedures that are part of the respective object. Thus clients are not troubled with implementation details and are not affected by later changes in the implementation of the object.

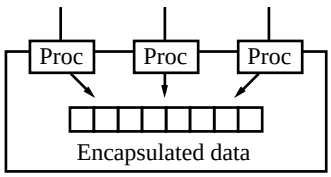

Fig. 1.3 An object with encapsulated data and a procedural interface

Information hiding was propagated by David Parnas [Par72]. It is not restricted to object-oriented languages but also supported by

numerous other modular languages such as Modula-2 with its *modules* and Ada with its *packages*.

Data abstraction

Data abstraction is the next step after information hiding. The objects described above exist only once, yet sometimes multiple copies of them are needed (Fig. 1.4).

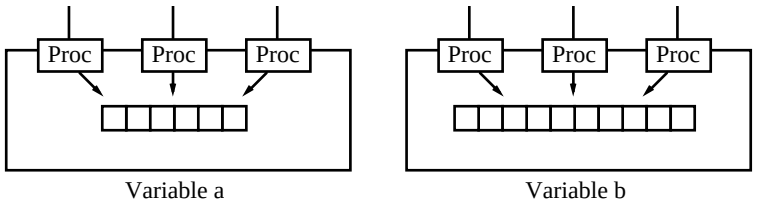

Fig. 1.4 Two variables *a* and *b* of an abstract data type

Just as we can declare any number of variables of a specific data type Integer, we want to be able to declare multiple variables of an abstract data type *Binary Tree*. As the operations +, -, * and DIV belong to *Integer*, a *Binary Tree* should provide operations such as insertion, deletion and searching for elements.

Integer	+, -, *, DIV, MOD, =, #, <, <=, >, >=
Binary tree	Insert, Delete, Search, Traverse, ...

An abstract data type is thus a unit consisting of data and the operations applicable to them. Multiple variables of such a type can be declared. Abstract data types are likewise not an invention of the object-oriented camp; they also can be realized in Modula-2 and Ada.

Inheritance

Inheritance is a concept not found in any conventional programming language. It means that an existing abstract data type can be extended to a new one that inherits all the data and operations of the existing type. The new type can include additional data and operations and can even modify inherited operations. This makes it possible to design a type as a semi-finished product, store it in a library, and later extend it to produce various final products (Fig. 1.5).

An important consequence of inheritance is that the extended type is compatible with the original one. All algorithms that work with objects of the original type can also work with objects of the new type. This greatly promotes the reusability of existing algorithms.

Dynamic binding

The fourth characteristic of object-oriented programming languages is *dynamic binding* of messages (requests) to procedures. When the message *Area* is sent to an object, the decision regarding which procedure is to carry out the request is made at run time, i.e., dynamically.

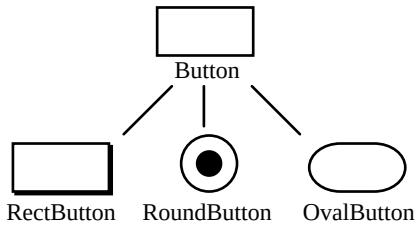

Fig. 1.5 A base type *Button* and various extensions

The compatibility between a type and its extensions makes it possible to store in a variable of type *T* not only objects of type *T*, but also objects of any extension of *T*. Thus a variable can be *polymorphic* (i.e., containing objects of multiple types). Depending on the type of the object stored in a variable at run time, messages are carried out differently. If variable *f* contains a *Rectangle* object, *f.Area* invokes the *Area* procedure for rectangles (Fig. 1.6 a); if *f* contains a *Circle* object, *f.Area* invokes the *Area* procedure for circles (Fig. 1.6 b).

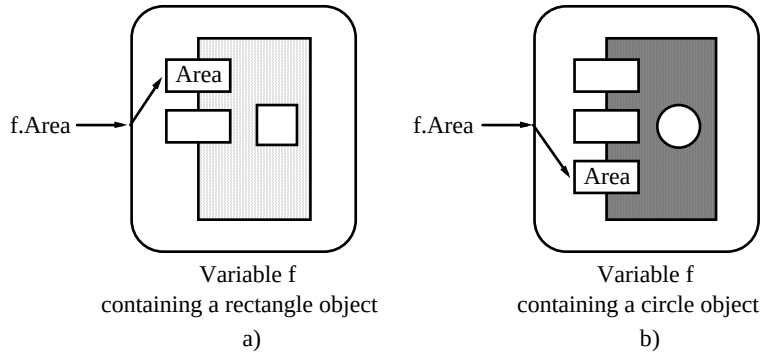

Fig. 1.6 Dynamic binding: the message *f.Area* is carried out by the *Area* procedure of the object that is stored in the variable *f* at run time

Dynamic binding has also been known for a long time in the form of procedure variables. The activation of a procedure variable causes the invocation of the procedure contained in it at run time. Working with procedure variables, however, is troublesome and

error-prone, while dynamic binding in object-oriented languages represents an elegant and reliable solution.

Extensible abstract data types with dynamically bound messages are called *classes*. Classes are the basic building blocks of object-oriented programming. They will be treated in detail beginning in Chapter 4. In summary we can say:

Object-oriented programming means programming with abstract data types (classes) using inheritance and dynamic binding.

1.4 How OOP Differs from Conventional Programming

Object-oriented terminology

Upon first contact with OOP, one immediately notices its unaccustomed terminology. We work with *classes* instead of data types, and we send *messages* instead of calling procedures. These terms were introduced in Smalltalk [GoR83], one of the first object-oriented languages, and have gained widespread acceptance despite the fact that (apart from subtle differences) conventional terminology would have sufficed.

Table 1.7 translates the most important terms of object-oriented languages into conventional terminology. The object-oriented terms are usually more concise and handier than their conventional counterparts. Therefore we will use them throughout this book. The reader should be aware, however, that these terms do not represent radically new concepts, but have their corresponding terms in conventional programming.

Object-oriented term	Conventional term
Class	Extensible abstract data type
Object	Instance of a class
Message	Procedure call (dynamically bound)
Method	Procedure of a class

Table 1.7 Object-oriented terminology

Another difference is the unaccustomed syntax of procedure calls in object-oriented languages. In order to invoke a procedure that draws a circle with a given *color* , we write:

```
circle.Draw(color)
```

We say that we send the message *Draw* to the object stored in *circle* (or simply to the object *circle*). The message merely represents a request rather than a procedure. It is the object that determines which procedure is to carry out the request. Because the object is the focus of attention, *circle* is written in front of the message name.

These differences, however, are of minor importance. Instead, the following properties are more essential:

- concentration on the data
- emphasis on reusability
- programming by extension
- distributed state and distributed responsibilities

Concentration on the data

Object-oriented programming focuses on the *objects* rather than on the procedures. In fact, there are programmers who insist that no procedure should exist that is not associated with some object. This goes too far, for there are certainly situations in which the algorithm bears more weight than the data. Nevertheless, data are usually the central points of object-oriented design around which the procedures crystallize.

Emphasis on reusability

Object-oriented design strives harder to achieve reusability than conventional design does. The goal of most conventional design methods, such as stepwise refinement [Wir71], is to find a customized solution to a specific problem. This results in tailored programs that are usually correct and efficient but very sensitive to changes in the requirements. Even a small change in the specifications could scrap the entire design.

In object-oriented design, the goal is not to tailor the classes to the clients, but rather to design the classes independently of their context and adapt the clients to the classes. One strives to make the classes more general than would be necessary for a specific application. This requires additional time during development, but pays off long term: the classes can be reused in other programs for which they were not originally designed.

Programming by extension

Object-oriented software is seldom written from scratch. OOP typically means extending existing software. Components such as windows, menus and switches are usually available as semifinished products in a library; they can be extended to meet specific requirements. Whole frameworks of classes can be taken from such libraries and extended to a complete program.

Distributed state and distributed responsibilities

In conventional programs the program state is stored in the global variables of the main program. Although the main program invokes procedures, the procedures usually do not have a state of their own, but either transform input data into output data, or work on global data (Fig. 1.8).

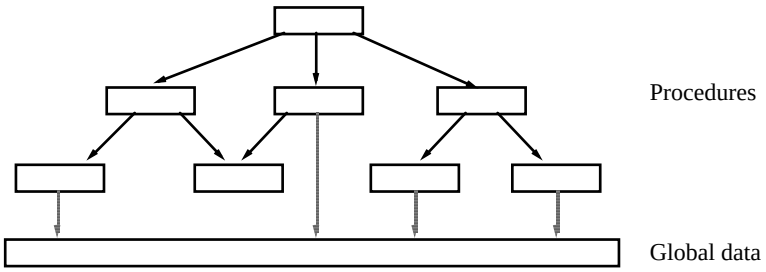

Fig. 1.8 Calling graph of procedures working on a set of global data

In object-oriented programs the state is distributed among multiple objects. Each object has its own state (its own data) and a set of procedures working on that state. The object is responsible not only for a single computation, but for a whole set of services.

Both state and responsibilities are more distributed in object-oriented programs. The main program and its global data are less important and often do not even exist. Objects communicate with one another in order to perform a specific task (Fig. 1.9). An object knows *what* other objects are responsible for, but does not know *how* they fulfill these responsibilities.

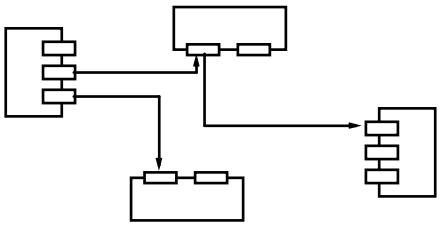

Fig. 1.9 Objects communicate by means of messages. Each object is responsible for a set of services that it provides to other objects.

Let us examine the example of a window system that, among other things, processes mouse clicks. An object-oriented window system registers such clicks, but it does not process them itself. It is not aware of the specific window types and hence does not know how they would react to mouse clicks (perhaps by positioning an

insertion point, by marking selected text, by drawing a figure, etc.). Thus it passes the click on to the respective window object and leaves further processing to that window. Processing mouse clicks is not the responsibility of the window system, but of the window in which the mouse key was pressed.

1.5 Classes as Abstraction Mechanisms

Semantic gap

Classes allow the modeling of real world entities. It is interesting to look at their history in programming languages. The driving force behind the introduction of classes was the striving for abstraction and the desire to bridge the *semantic gap* between problem-oriented specifications and machine-oriented programs.

Initially, abstractions for data and for operations were developed independently, but for some years there has been the tendency to combine them. Object-oriented programming is a consequence of this trend (Fig. 1.10).

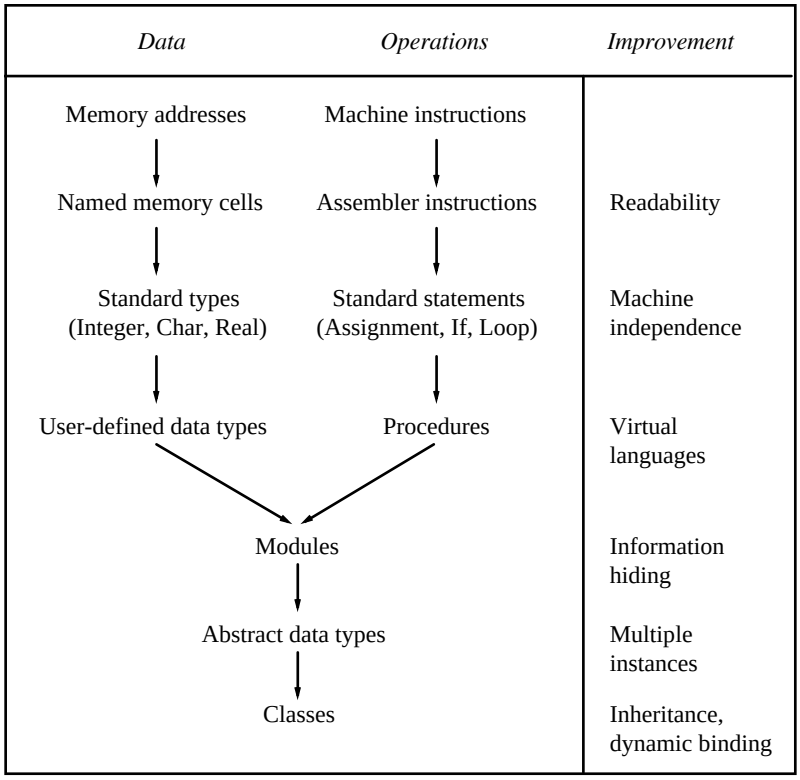

Fig. 1.10 The development of abstractions in programming languages

The earliest programs used memory cells as data and machine instructions as operations. The greatest problem in programming was to map real-world entities such as a customer or an account onto the machine level. There was a broad gap between the problem domain and the program.

Variable names and mnemonic operation names

The first improvement came with assemblers, which made it possible to give memory cells a name and a primitive structure, and replaced binary instruction codes with mnemonic instruction names. This improved the readability of programs, but contributed little to reducing the semantic gap.

Standard types and standard operations

Only with the advent of higher programming languages such as Fortran could the semantic gap be reduced. Now arithmetic expressions could be written in common mathematical notation rather than needing to be reduced to a sequence of machine instructions. The first simple data types such as Integer and Real were introduced along with a set of operations that could be applied to them. Although the data types and operations were

determined by the programming language, the machine independence that was achieved represented immense progress in the level of abstraction.

Note that Integer has almost all the properties of an abstract data type. Users of Integer variables do not need to know whether the most significant bit is left or right, or which machine instructions are used to realize the + operation. The only difference with regard to abstract data types is that Integer is built into the programming language, while abstract data types are defined by the programmer.

User-defined data types and procedures

In the 1960s languages like Pascal were developed that allowed the programmer to create a virtual language. One no longer had to restrict oneself to the data types and operations of a particular language; instead, the programmer could define custom data types and custom operations in the form of procedures. The resulting virtual language was tailored to a specific problem domain and thus more problem-oriented than a concrete language.

Modules

So far, data and operations had developed separately, although it is interesting to note that similar developments took place almost simultaneously in the two branches. First they achieved better readability, then machine independence, and finally more problem orientation. At the end of the 1970s it was recognized that data and their associated operations should be combined into modules. This brought more order into programs. Being collections of data and operations, modules are better suited to modeling real-world entities than procedures are. Modules are taken for granted in most modern programming languages; without them the development of large program systems would be much more complicated and error-prone.

Abstract data types

The problem with modules is that there is only one instance of them. If multiple copies are needed, one has to use abstract data types which, like modules, consist of data and operations, but can be used to declare several variables of this type. Abstract data types already existed in languages such as Modula-2 and Ada.

Classes

Object-oriented languages introduced the concept of classes. Classes are abstract data types supporting inheritance and dynamic binding. They are perfectly suitable to modeling real-world entities such as sensors, switches, or displays in software. The semantic gap between the problem domain and the program nearly disappears.

1.6 History of Object-Oriented Languages

Object-oriented programming is by no means new. The term was coined in the early 1970s in connection with Smalltalk [GoR83], a programming language developed by a research group at Xerox PARC. The roots of OOP, however, go back even farther to the language Simula [BDMN79], which was developed at the University of Oslo in 1967. Simula already had, in essence, all the properties of today's object-oriented languages. Thus OOP was around already a quarter of a century ago, which makes it even more surprising that the approach only recently began to gain widespread acceptance. This probably stems from the fact that Simula and Smalltalk were considered specialized languages: Simula was designed as a simulation language, and Smalltalk was viewed as a toy by many computer scientists. The value of classes for general programming was recognized only later.

Smalltalk

Smalltalk became the prototype of object-oriented languages. It still is one of the most consistent OOP languages, for all its data types are classes and all its operations are messages. Smalltalk is usually interpreted, making its execution slow. Although newer Smalltalk systems do generate machine code, message dispatching is still interpretative. Furthermore, Smalltalk does not allow static type checking. This limits its suitability for larger software systems.

Hybrid languages

In the mid-1980s many new object-oriented languages emerged; most of them were *hybrid* in nature and were extensions of existing languages such as Pascal and C. Hybrid languages include conventional data types (such as Integer and arrays) in addition to classes, and procedures in addition to messages. These languages permit type checking at compile time. Programs are translated into machine code, improving their efficiency over interpreted systems. The ease of switching from a familiar language like Pascal to a dialect like Object-Pascal [Sch86] contributed to the acceptance of such extensions and thus of object-oriented programming in general. Object-oriented dialects are now available for a wide range of languages.

Oberon-2

Oberon-2, the language used in this book, is also a hybrid. In fact, Oberon-2 is even closer to conventional languages because it does not have a special class construct: Classes are simply records that contain procedures in addition to data.

1.7 Summary

The most significant properties of OOP are the following:

(1) Data and operations are combined into classes that serve as types for objects.
(2) Classes can be extended to create new classes containing additional data and operations. Objects of an extended class can be used wherever objects of their base class are permitted.
(3) Operations on objects are usually not performed by procedure calls; instead, objects are sent messages. A message is a request, and it is up to the receiving object to determine which procedure is to handle the request. Objects communicating via messages are more loosely coupled than software components statically connected via procedure calls.

2 Oberon-2

Throughout this book we will use the programming language Oberon-2, an object-oriented language in the tradition of Pascal and Modula-2

This chapter introduces the reader to Oberon-2. We do not provide an introduction to programming, but assume that the reader is already able to read and write programs. Anyone who understands Pascal or, better yet, Modula-2 can read Oberon-2 programs without difficulty. Thus Oberon-2 is only described informally on the basis of several examples. Answers to more detailed questions can be found in the language definition in Appendix A.

Oberon-2 evolved from Oberon, which, like its predecessors Pascal and Modula-2, was developed by *Niklaus Wirth* [ReW92]. Several features of Modula-2 such as variant records, enumeration types and subrange types were omitted in Oberon. The language concentrates on the essentials and is thus well suited for both education and practice. New features in Oberon include the concept of type extension (inheritance); Oberon-2 finally adds type-bound procedures (methods).

Oberon is not only a programming language, but also an operating system that provides a run-time environment with command activation, garbage collection, dynamic loading of modules, and certain run-time data structures [Rei91, WiG92; see also Appendix A.12.4]. In Oberon the language is interwoven with the operating system. For the user to fully enjoy the power of Oberon, the language needs to be combined with the Oberon System, under which both Oberon and Oberon-2 programs run.

2.1 Features of Oberon-2

Oberon-2's most important features are block structure, modularity, separate compilation, strong type checking at compile time, type extension, and type-bound procedures.

Block structure allows nested procedures with separate scopes for identifiers. *Modules* permit the decomposition of large programs into smaller, comprehensible parts that can be compiled separately. The compiler ensures that their interfaces match. This is called *separate* compilation to distinguish it from *independent* compilation, in which no interface checking takes place (such as in Fortran or C).

Strong type checking means that the compiler checks at every operation (assignment, arithmetic, relational, etc.) whether variables are used according to their declaration and hence according to the intentions of the programmer. In this way many errors can already be detected at compile time, which drastically reduces the cost of corrections.

The object-oriented features of Oberon-2 are not yet treated in this chapter. They are described in Chapters 4 to 6 and then used extensively throughout the rest of this book.

2.2 Declarations

Data types

All identifiers appearing in a program (i.e., all names of constants, types, variables, and procedures) must be declared before they are used. In their declaration they are assigned a data type. Oberon-2 has basic types and composite types. The basic types are listed in Table 2.1.

	Type name	Typical range
Integer numbers	SHORTINT	-128..127
	INTEGER	-32768..32767
	LONGINT	-2147483648..2147483647
Real numbers	REAL	±3.40282E38 (4 bytes)
	LONGREAL	±1.79769D308 (8 bytes)
ASCII characters	CHAR	0X..0FFX (0..255 hexadecimal)
Boolean values	BOOLEAN	TRUE, FALSE
Sets	SET	Sets of numbers in the range 0..31

Table 2.1 Basic types in Oberon-2

The ranges of the basic types are not defined by the language. On most machines, however, the values given in the right column of Fig. 2.1 apply. Composite data types are *arrays*, *records*, *pointers* and *procedure types*.

Arrays

An *array* is a collection of elements all of the same type (the element type). The elements do not have individual names, but are selected via an index. Examples of array variables are:

```
VAR
 a: ARRAY 10 OF CHAR;  (* a has 10 elements: a[0], …, a[9] *)
 b: ARRAY 100, 100 OF INTEGER;
```

Arrays are indexed with integers, the first element having the index 0. The elements are referenced as $a[i]$ and $b[i, j]$, whereby the index values are checked to assure that they are within the declared range.

Records

A *record* is a collection of named *fields* of arbitrary type, for example:

```
TYPE
 Person = RECORD
    name: ARRAY 32 OF CHAR;
    idNumber: INTEGER;
    salary: REAL
 END;
```

IF *r* is a variable of type *Person*, its fields can be referenced as *r.name*, *r.idNumber* and *r.salary*. Records can be extended to create new types (see Chapter 5).

Pointers

A *pointer variable* contains the address of a record or an array, or it has the value NIL, which means that it does not point to any record or array. Examples of pointer types are:

```
TYPE
 PersonPtr = POINTER TO Person;
 Box = POINTER TO RECORD x, y, width, height: INTEGER END;
 Vector = POINTER TO ARRAY 100 OF INTEGER;
 String = POINTER TO ARRAY OF CHAR;
```

If *p* is a variable of type *PersonPtr*, then *p*^ is the (nameless) record of type *Person* (the pointer base type) to which *p* points. The field *name* is referenced with *p*^*.name*. For the sake of simplicity the symbol ^ can be omitted, leaving *p.name*. This is an abstraction from the fact that *p* is only a pointer to a record and not the record itself. However, one must be aware that in the assignment *q*:=*p* only the pointer *p* is assigned and not the record *p*^. The

invocation of the predeclared procedure NEW(*p*) allocates memory for *p*^.

If *s* is a variable of type *String*, then *s*^ is the array to which *s* points. The array has been declared without length and is thus called an *open array*. Its length is specified at run time. *s*^[*i*] denotes the element with index *i*. Here, too, the symbol ^ can be omitted, leaving *s*[*i*]. NEW(*s*, *n*) allocates memory for the array *s*^ with *n* elements.

In Oberon, dynamically allocated memory is never explicitly deallocated. Instead, the Oberon System features a *garbage collector* that collects and recycles regions of memory that are no longer referenced by a pointer. This resolves a frequent source of errors: The programmer could deallocate memory to which some pointer still refers. Dereferencing via such a dangling pointer would lead to an error.

Procedure types

Variables of type procedure (*procedure variables*) contain as their value either a procedure or NIL (no procedure). When a procedure variable is invoked, the procedure currently stored in it is activated. In the following example the procedure *WriteTerminal* is assigned to the procedure variable *write*:

```
VAR write: PROCEDURE (ch: CHAR);

PROCEDURE WriteTerminal (ch: CHAR);
BEGIN …
END WriteTerminal;

write := WriteTerminal;
write(ch);   (*activates WriteTerminal*)
```

2.3 Expressions

Expressions describe the computation of values and consist of operators and operands. There are four kinds of expressions, which are shown in Table 2.2.

	Operators	Result type
Arithmetic expressions	+, -, *, /, DIV, MOD	Numeric
Boolean expressions	&, OR, ~	BOOLEAN
Relational expressions	=, #, <, <=, >, >=, IN	BOOLEAN
Set expressions	+, -, *, /	SET

Table 2.2 Kinds of expressions in Oberon-2

Arithmetic expressions and relational expressions

The meaning of the arithmetic and relational operators is obvious. It should be noted, however, that the compatibility rules in Oberon-2 are less restrictive than in Pascal or Modula-2. In particular, numeric types (INTEGER, REAL, etc.) can be mixed in arithmetic expressions, and character arrays can be compared. The following examples will answer most questions. The detailed compatibility rules can be found in the language definition in Appendix A.

```
VAR
 i: INTEGER; j: LONGINT; r: REAL;
 set: SET;
 s: ARRAY 32 OF CHAR;
 sp: POINTER TO ARRAY OF CHAR;
 p, p1: PersonPtr;  (*see declaration in previous section*)
 proc: PROCEDURE (x: INTEGER);
```

Expression	*Result type*
3	SHORTINT
300	INTEGER
100000	LONGINT
0X	CHAR
i + j	LONGINT
i + 3*(r-j)	REAL
i DIV j	LONGINT
i / j	REAL
(s > "John") OR (s = sp^)	BOOLEAN
s = "a"	BOOLEAN
p # p1	BOOLEAN
proc = NIL	BOOLEAN
~ (i IN set)	BOOLEAN

Boolean expressions

The expression *~x* means the negation of *x*. The operators & and OR are not commutative and are evaluated as follows:

```
a & b        if a then b else false end
a OR b       if a then true else b end
```

This is called *short circuit evaluation* because the evaluation of the expression stops as soon as its value is known; this proves especially useful for expressions like the following:

```
IF (p # NIL) & (p.name = "John") THEN … END
```

If *p* = NIL, the second part of the expression is not evaluated; thus improper dereferencing of *p* is avoided.

Set expressions

The set operators have the following meanings:

+	Union	{0..7} + {5..9} = {0..9}
-	Difference (x-y = x*(-y))	{0..7} - {5..9} = {0..4}
*	Intersection	{0..7} * {5..9} = {5..7}
/	Symmetric difference (x/y = (x-y)+(y-x))	{0..7} / {5..9} = {0..4, 8..9}

The expression *i IN s* tests whether the number *i* is contained in the set *s*.

2.4 Statements

Oberon-2 provides elementary statements (assignment, procedure call, return, exit), as well as structured statements for selection (if, case) and iteration (while, repeat, for, loop). The meanings of these statements are so common that the following examples should suffice. The reader can find details as well as the meanings of the predeclared procedures (ORD, CHR, etc.) in the language definition (Appendix A).

```
p.name := "John"                                    (*assignment*)
i := 10*i + ORD(ch)-ORD("0")

WriteInt(i, 10)                                     (*procedure call*)
i := Length(text)

r := p MOD q;                                       (*while*)
WHILE r # 0 DO
    p := q; q := r; r := p MOD q
END

i := 0;                                             (*repeat*)
REPEAT
    s[i] := CHR(ORD("0") + n MOD 10);
    n := n DIV 10;
    INC(i)
UNTIL n = 0

FOR i := 0 TO LEN(s)-1 DO s[i] := 0X END            (*for*)

i := 0;                                             (*loop, exit, if, return*)
LOOP
    ReadChar(ch);
    IF i = LEN(s) THEN Error; RETURN
    ELSIF ch = 0X THEN EXIT
    END;
    s[i] := ch; INC(i)
END
```

```
CASE ch OF                                    (*case*)
    "a".."z", "A"..."Z": ReadIdentifier
|   "0".."9": ReadNumber
|   " ' ", ' " ': ReadString
ELSE ReadSpecial
END
```

Note that string constants can be assigned to a character array of fixed length as long as the array is sufficiently long to hold the string and the terminal character 0X that is automatically inserted during the assignment.

Also note that every structured statement ends with a keyword (usually END) and may contain a whole sequence of statements. Contrary to Pascal, the statement sequence need not be bracketed in BEGIN … END.

2.5 Procedures

For procedures, an example will also suffice. The procedure below converts a number *n* to a character array *hex* that represents the hexadecimal representation of the number.

```
PROCEDURE IntToHex (n: LONGINT; VAR hex: ARRAY OF CHAR);
 VAR i, k: INTEGER; s: ARRAY 8 OF CHAR;

 PROCEDURE Hex (i: LONGINT): CHAR;
 BEGIN (*0 <= i <= 15*)
    IF i < 10 THEN RETURN CHR(i + ORD("0"))
    ELSE RETURN CHR(i-10 + ORD("A"))
    END
 END Hex;

BEGIN (*IntToHex: assumes n >= 0*)
 i := 0;
 REPEAT s[i] := Hex(n MOD 16); INC(i); n := n DIV 16 UNTIL n = 0;
 k := 0;
 REPEAT DEC(i); hex[k] := s[i]; INC(k) UNTIL i = 0;
 hex[k] := 0X
END IntToHex;
```

Procedures consist of a declaration part, in which constants, types, variables and further procedures can be declared locally, and a statement part (the body), which is executed when the procedure is invoked. The parameters declared in the procedure heading (*n* and *hex*) are called *formal* parameters. They are considered local to the procedure. The parameters specified at the procedure call are termed *actual* parameters.

Scope

The *scope* of an identifier, i.e., the range in which the identifier can be used, extends textually from its declaration to the end of the block (procedure or module) in which it is declared. It overrides the scope of any identically named identifier declared in an outer block. The scope of the parameter *i* in *Hex* overrides the scope of the variable *i* in *IntToHex*. Nested scopes allow the declaration of arbitrary identifiers in every procedure without having to bother about whether an identifier was already declared outside the procedure. Good programming style suggests that a procedure work only with its own local variables (including its parameters) and that it not use global variables or—even worse—local variables of an enclosing procedure.

Parameters

In the procedure *IntToHex*, *hex* is called a *variable parameter* because it is declared with the symbol VAR. A variable parameter has the same address as its corresponding actual parameter, which must be a variable. Thus if *hex* is modified in the procedure, the actual parameter is modified, too. Variable parameters are used as output parameters.

n is a *value parameter* because during the procedure invocation the value of the actual parameter is assigned to *n*. Thus *n* contains a local copy of the actual parameter. Changing the value of *n* does not affect the value of the actual parameter. Value parameters are used as input parameters.

hex is an *open array* parameter. Its length is determined at run time and is equal to the length of the actual parameter, which must likewise be an array.

Function procedures

IntToHex is a procedure that is invoked as a statement. *Hex*, on the other hand, is a function procedure that is invoked as part of an expression. It returns a value that is used in the evaluation of the expression. The value that a function procedure is to return must be specified in a return statement. A function procedure is characterized by the declaration of a result type following its formal parameter list.

Recursion

Procedures can invoke themselves recursively. With each invocation, a new set of local variables is allocated, so that every invocation of the procedure works with its own local variables.

Standard procedures

A number of standard procedures such as ORD, CHR, LEN and COPY are predeclared. Their descriptions are given in Appendix A.10.3.

2.6 Modules

Large programs are normally decomposed into smaller units, called modules. A compiler, for example, consists of a scanner, a parser, a code generator, and a table handler (Fig. 2.3). Each of these modules works on a well-defined subdomain of the problem and is easier to understand than the compiler as a whole.

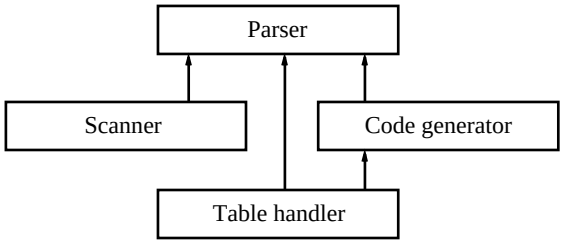

Fig. 2.3 Modules of a compilers: Arrows indicate the "is used by" relationship

The module interface

A module is a unit with a clearly defined *interface;* it can be used without knowledge of how it is implemented, and it can be implemented without knowledge of the context in which it might later be used.

In line with this definition, a module in Oberon-2 is a collection of constants, types, variables, and procedures that form a logical and syntactical entity. Its interface consists of the declarations of the identifiers that can be used by other modules. The module is said to *export* these identifiers.

Let us take the example of a module that represents the implementation of a dictionary in which word pairs can be entered and looked up. The first word serves as the key and the second as the value.

When designing a module like this, we first define its interface by writing a skeletal module consisting only of the declarations of the exported identifiers. For the dictionary this could be:

```
MODULE Dictionary;
 TYPE String* = ARRAY 32 OF CHAR;
 PROCEDURE Clear*; END Clear;
 PROCEDURE Enter* (key, value: String); END Enter;
 PROCEDURE Lookup* (key: String; VAR value: String); END Lookup;
 PROCEDURE Print*; END Print;
END Dictionary.
```

An identifier is marked as exported by adding an asterisk (*) after its name in its declarations. Thus the module exports the type *String*, as well as the procedures *Clear* for erasing the dictionary, *Enter* for entering a new word pair, *Lookup* to search for a word pair with a given key, and *Print* to output the dictionary on the terminal.

Implementation

This skeleton is later complemented by further declarations and statements until the implementation is complete. Now let us tackle the full implementation of module *Dictionary*. For the sake of simplicity, it is implemented using an unsorted linked list.

```
MODULE Dictionary;
IMPORT IO;

TYPE
 String* = ARRAY 32 OF CHAR;
 Node = POINTER TO NodeDesc;
 NodeDesc = RECORD
     key, value: String;
     next: Node
 END;

VAR root: Node;

PROCEDURE Clear*;
BEGIN root := NIL
END Clear;

PROCEDURE Enter* (key, value: String);
 VAR p: Node;
BEGIN
 NEW(p); p.next := root; root := p; p.key := key; p.value := value
END Enter;

PROCEDURE Lookup* (key: String; VAR value: String);
 VAR p: Node;
BEGIN p := root;
 WHILE (p # NIL) & (p.key # key) DO p := p.next END;
 IF p # NIL THEN value := p.value ELSE value := "" END
END Lookup;

PROCEDURE Print*;
 VAR p: Node;
BEGIN p := root;
 WHILE p # NIL DO
     IO.Str(p.key); IO.Str("   "); IO.Str(p.value); IO.NL;
     p := p.next
 END
END Print;

BEGIN Clear
END Dictionary.
```

Note that the types *Node* and *NodeDesc* as well as the variable *root* are declared without an export mark and are therefore not visible outside *Dictionary*.

The interface of the module can be extracted from the implementation at any time using a *browser* (see Section 2.6). The browser simply collects the declarations of all exported identifiers and shows them in the following form:

```
DEFINITION Dictionary;
 TYPE String = ARRAY 32 OF CHAR;
 PROCEDURE Clear;
 PROCEDURE Enter (key, value: String);
 PROCEDURE Lookup (key: String; VAR value: String);
 PROCEDURE Print;
END Dictionary.
```

Note that this is not an Oberon-2 module, but only a special view of it, namely its interface. In languages like Modula-2 the programmer has to write the interface description (the definition module) manually and separately from the implementation of the module. Consistency between the two documents must be maintained manually. In Oberon-2 there is only one document per module: the implementation. The interface is only a special view on it; it is extracted automatically and therefore always consistent with the implementation. This is significant progress over the Modula-2 approach.

Import

For the output of words, *Dictionary* uses the module *IO*, which is *imported* at the beginning of *Dictionary*. *IO*'s interface looks like this:

```
DEFINITION IO;
 PROCEDURE Str (s: ARRAY OF CHAR);
 PROCEDURE Int (i: LONGINT; w: INTEGER);
 PROCEDURE NL;  (*skip to next line*)
 …
END IO.
```

All identifiers exported by *IO* can be used in *Dictionary* or any other module that imports *IO*. They only need to be *qualified* with the name of the exporting module. The procedure *Dictionary.Print* contains invocations of *IO.Str* and *IO.NL*, for example.

Separate compilation with interface checking

An important feature of Oberon-2 is that the compiler checks the correct use of interfaces. When a module is compiled, a description of its interface is written to a *symbol file* in machine-readable form. During the compilation of a client module, the compiler obtains the symbol files of the imported modules and

thus knows the identifiers exported by those modules as well as their types. This permits type checking as if the exported identifiers had been declared in the importing module itself.

This is called *separate compilation*, in contrast to *independent compilation*, in which modules can be individually compiled, but the compiler does not check the interfaces.

During the compilation of *Dictionary* the compiler checks whether *IO* is used in accordance with its interface. If this interface happens to be modified later, then the previous check is no longer valid, and *Dictionary* has to be recompiled to recheck the correct use of *IO*. Meanwhile the operating system makes sure that *Dictionary* cannot be executed until it has been recompiled. Thus any modification in the interface of a module *M* would also require the recompilation of all its *clients* (all modules that import *M*).

Module body

In addition to its procedures a module can also contain its own code. This code is called the *module body* (the statement sequence at the end of *Dictionary*). The module body primarily serves to initialize the global data of the module. It is executed as soon as the module is loaded. Prior to that, however, the bodies of all imported modules are executed. (The imported modules need to be initialized before the importing module, or they could not be used in the body of the importing module.) This means that Oberon-2 does not permit cyclic import relationships among modules. The initialization sequence would otherwise be undefined.

Read-only export

A variable or a record field can be exported as read-only so that clients can read its data, but they cannot make modifications. This increases the reliability of the system, because the exporting module can be sure that clients will not destroy its data. Read-only variables and fields are marked with a minus sign (-) instead of an asterisk (*) in their declaration. A file system, for example, could write-protect its data as follows:

```
MODULE FileSystem;
TYPE
 File* = POINTER TO FileDesc;
 FileDesc* = RECORD
    name-: ARRAY 32 OF CHAR;
    length-: LONGINT;
    …
 END;

VAR resultCode-: INTEGER;
…
```

```
END FileSystem.
```

The fields *name* and *length* as well as the variable *resultCode* can be read but not modified by clients. Only the exporting module *FileSystem* can modify them, because they are declared in that module. If a structured variable is read-only, this also applies to its components. Not only *file.name* but also *file.name*[*i*] is read-only.

In languages such as Modula-2, data that are not to be modified must be made available via access procedures. Read-only export is a more efficient solution.

The purpose of modules

What are modules for? First of all, they are a *structuring medium*. They group data and their associated operations together and help to create order in a program.

Modules are also an *abstraction medium*. They hide implementation details from other modules and provide their services via a simple interface. A module forms a wall. Identifiers declared within a module are visible outside only if they are exported. Identifiers exported by a module *A* are visible within a module *B* only if *A* is imported by *B*. Import and export make the coupling between modules visible.

Finally, a module is a *compilation unit*. Its source code is stored in a file and the resulting object code is written to another file. Thus modules are the smallest interchangeable components in a system. The code generator in Fig. 2.3 can be replaced with another one without recompilation, but not an individual procedure of the code generator.

2.7 Commands

The explanations thus far referred to the *language* Oberon-2; this section treats features of the Oberon *operating system*.

In most operating systems the smallest units that can be invoked in dialog with the computer are programs. In the Oberon System these units are *commands*. A command is any parameterless procedure *P* that is exported by a module *M*. In a typical Oberon environment a command is activated by typing its name (*M.P*) in a window and clicking it with the middle mouse button. Usually the name of the command is already displayed in some window and only needs to be clicked.

When the command *M.P* is activated, the module *M* and all modules imported by *M* are loaded (if they are not already in

memory) and the procedure *P* is executed. After *P* terminates, *M* remains loaded with all its global data and their values. If *M.P* (or another command from *M*) is invoked again, *M* is not loaded anew. *P* finds the values of the global data just as they were left after *P*'s last invocation.

Commands can thus communicate with one another via data structures in main memory rather than via files. This is simpler and more efficient and makes it possible to hide the data structure within the module to which the commands belong.

Let us now rewrite the *Dictionary* example of Section 2.6 so that *Clear*, *Enter*, *Lookup* and *Print* can be invoked as commands by the user. The interface of *Dictionary* would look like this:

```
DEFINITION Dictionary;
 PROCEDURE Clear;
 PROCEDURE Enter;
 PROCEDURE Lookup;
 PROCEDURE Print;
END Dictionary.
```

All four procedures are now commands and can be invoked like programs. But how do they obtain their arguments?

Command arguments

Each command can decide itself what kind of data it accepts as arguments: the text following the command name, the text in the current selection, the text at the insertion point, or some other marked object on the screen. The Oberon System provides appropriate procedures to read such arguments.

In our example, we obtain the arguments from the text following the command. The user activates the commands as follows:

```
Dictionary.Enter      book Buch
Dictionary.Lookup   book
```

Enter takes the word pair "book Buch" as its parameter and enters it in the dictionary. *Lookup* takes the word "book", searches for it in the dictionary, and returns the word "Buch". The command *Enter* is implemented as follows:

```
PROCEDURE Enter*;  (*read two words following the command text*)
 VAR s: IO.Scanner; p: Node;
BEGIN NEW(p);
 s.SetToParameters; s.Read; (*read first word*)
 IF s.class = IO.name THEN
    COPY(s.str, p.key ); s.Read; (*read second word*)
    IF s.class = IO.name THEN
       COPY(s.str, p.value);
       p.next := root; root := p  (*link p to the dictionary*)
    END
```

```
    END
END Enter;
```

IO.Scanner is a data type that allows convenient reading of names, numbers, characters, and strings (see Appendix C). The Scanner variable *s* is set to the text immediately following the command name by *s.SetToParameters*. *s.Read* reads the next symbol. Thus the command obtains its arguments and can proceed as in Section 2.6.

Note that *Dictionary* remains loaded after *Enter* terminates and the data of the dictionary thus retain their values. Succeeding invocations of *Enter* permit additional words to be entered, and with *Lookup* words can be searched for.

Unloading modules

When is *Dictionary* removed from memory? Oberon's solution is that modules must be explicitly unloaded on user demand. The Oberon System provides a command for that purpose. After *Dictionary* is unloaded, a new version of it can be loaded.

Linking loader

It should be noted that Oberon has a *linking loader* that links object modules with other modules only upon loading. There are no prelinked object files; rather, each object module is its own file.

The loader also makes sure that each module is in memory *only once*. If module *A* is loaded that imports an already loaded module *B*, then *A* is linked to the loaded *B* and *B* is not loaded anew. Since modules remain in memory after being loaded the first time, modules seldom have to be afterloaded. This reduces loading time and memory requirements for Oberon programs.

Purpose of commands

Commands are a useful language construct. They permit the creation of programs with multiple entry points. Commands can be invoked interactively without needing a main program. They prove especially practical in the creation of large systems consisting of several equally important services, such as an electronic mail system with services such as sending mail, reading mail, deleting mail, etc. Which of these services should become the main program and which should be subordinate? Commands allow offering all these services on the same level without the need to create an artificial superordinate main program.

3 Data Abstraction

Abstraction is the most effective weapon against complexity. It means concentrating on the essentials and ignoring the details. Large systems can only be made comprehensible by decomposing them into modules that are simple from the outside and hide all complexity within.

The principle of abstraction has been successfully applied to many technical things; e.g., anybody can operate a television set without understanding the circuitry within the device. The same should also apply to software. We strive for modules with simple interfaces that can be used without knowing their implementation. In other words, we want to abstract from *concrete* data structures and attain *abstract* data structures, or, even better, abstract data types or classes.

3.1 Concrete Data Structures

In older programming languages like Pascal, all data structures are visible. A programmer can define custom data types, yet their structure is known to other parts of the program; indeed, the structure must be known in order for the programmer to work with these data. We call these *concrete data structures*.

Concrete data structure for a priority queue

Let us consider an example of a nontrivial data structure, a *priority queue*, to which elements can be added in any order and then retrieved in the order of their priority. For the sake of simplicity, we assume that the elements are numbers that simultaneously express their priority (smaller numbers representing higher priority). One efficient data structure for the implementation of priority queues is the *heap* [Sed88]. A heap is a binary tree with n elements that are arranged in the tree such that the value of the parent is always less than or equal to the value of its two

children. The tree is almost balanced: there exists a number *h* such that all nodes have height *h* or *h-1* (Fig. 3.1).

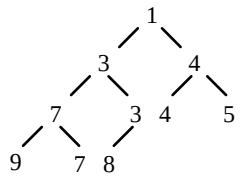

Fig. 3.1 Heap with 10 elements

Contrary to binary trees, there is no ordering between the two children of a node. The value of the left child could be smaller than, equal to, or greater than the value of the right child. However, the value of the parent is always less than or equal to the value of the children, which means that the root has the smallest value of the entire structure.

Fig. 3.1 shows that all levels of the tree except the last one are completely filled. The first level contains the element 1; the second level the elements 3 and 4; the third level 7, 3, 4 and 5; and so on. If the elements are stored in this sequence, an array can be used as concrete data structure, as shown in Fig. 3.2.

	1	2	3	4	5	6	7	8	9	10
a	1	3	4	7	3	4	5	9	7	8

Fig. 3.2 Array representation of the heap in Fig. 3.1

The advantage of this implementation is that pointers do not need to be stored; the children of elements *a[i]* (if they exist) are located at *a[2*i]* and *a[2*i+1]*. For a given element *a[i]*, the parent (if it exists) is located at *a[i DIV 2]*. The concrete data structure of a heap that can hold up to 127 numbers takes the following form:

```
VAR
 a: ARRAY 128 OF INTEGER;
 n: INTEGER; (*number of elements in the heap*)
```

A new element is inserted by storing it at the end of the heap (in *a[n+1]*) and then swapping places with its parent (propagating it upward) as long as the value of the new element is less than the value of its parent. As Fig. 3.3 shows, the number of swaps is of order O(log *n*).

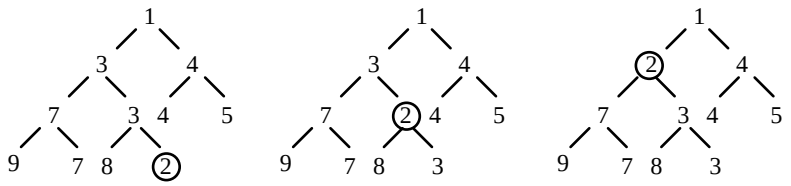

Fig. 3.3 Element 2 is appended to the end of the heap and then moved upwards until its parent element is of lesser or equal value

The following statements insert element *x* in heap *a* (we assume that the value MIN(INTEGER) is stored in *a[0]* as a sentinel):

```
(*virtually insert x at a[n]*)
n := n + 1;
(*propagate x from a[n] upwards*)
i := n;
WHILE x < a[i DIV 2] DO
 a[i] := a[i DIV 2]; i := i DIV 2
END;
a[i] := x
```

A heap is used in situations that require elements to be removed from a set in ascending order of value beginning with the smallest. A typical example is a set of processes that are to be ordered according to time or priority.

The smallest element is always located at *a[1]*. When it is removed, the heap must be adjusted. This is done by moving the last element *a[n]* to *a[1]* and then swapping places with the smaller of its children (propagating it downward) as long as it is larger than (both) its child(ren) (Fig. 3.4).

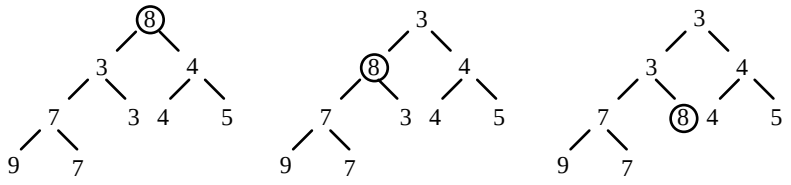

Fig. 3.4 Element *a[1]* was removed. Element *a[n]* = 8 was moved to *a[1]* and is now propagated downward in the tree

The following code segment removes the smallest element *x* from heap *a*:

```
x := a[1];
(*propagate a[n] from a[1] downwards*)
y := a[n]; n := n - 1; i := 1; ready := FALSE;
WHILE (i <= n DIV 2) & ~ ready DO
 j := i + i;
 IF (j < n) & (a[j] > a[j+1]) THEN j := j + 1 END;  (*select smaller child*)
 IF y > a[j] THEN a[i] := a[j]; i := j ELSE ready := TRUE END
END;
a[i] := y
```

Heap *a* and its number of elements *n* make up the concrete data structure for the priority queue. Clients can access the concrete data structure directly, but this is not recommended because of the following problems:

Clients are bothered with details

Clients must be familiar with both the declaration of the data structure and the algorithms for inserting and removing elements. This complicates working with the data and bothers clients with unnecessary details. The same code for accessing the data is often present in every module that uses the data, thus leading to duplication of code. Finally, clients may inadvartently destroy the consistency of the data (the heap order).

Modifications in the data affect the clients

Working with concrete data structures further has the disadvantage that modifications in the data affect the clients. If the implementation of the heap is changed from a fixed-length array to a tree in order to allow an arbitrary number of elements to be stored in it, then the access algorithms also change and all clients must be adapted. This is unpleasant because it requires knowing all locations where the data structure is used. It is easy to miss one.

The clients actually do not care how the priority queue is implemented. They simply want to use it as a black box. More important, they do not want to be affected by changes in its implementation. The concrete data structure thus needs to be hidden.

3.2 Abstract Data Structures

An abstract data structure is a unit consisting of data and procedures. The data are hidden within the unit and can only be accessed by means of dedicated procedures (Fig. 3.5). The data structure is termed abstract because only its name and its interface, but not its implementation, are known.

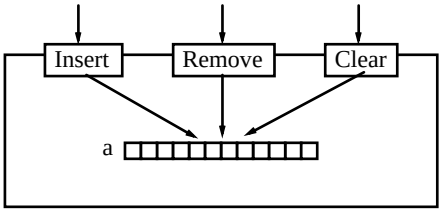

Fig. 3.5 Abstract data structure:
Heap *a* is accessible only via dedicated procedures

Information hiding

Abstract data structures support information hiding [Par72]. Their implementation is hidden behind an interface that remains unchanged, even if the implementation changes.

State

Abstract data structures have a *state* that can be modified by means of access procedures. The state is expressed in the values of the data structure and serves to store values between successive procedure invocations.

In Oberon-2, abstract data structures are implemented as *modules* that hide the data from clients by not exporting them. The priority queue thus becomes the module *PriorityQueue* with the following interface:

```
DEFINITION PriorityQueue;
 VAR n-: INTEGER;  (*number of elements*)
 PROCEDURE Insert (x: INTEGER);
 PROCEDURE Remove (VAR x: INTEGER);
 PROCEDURE Clear;
END PriorityQueue.
```

The module's three procedures *Insert* an element, *Remove* the smallest element, and *Clear* the queue, respectively. The number of elements is not provided by an access procedure, but directly as variable *n*. It is unlikely that its implementation will change, thus its type need not be hidden behind an access procedure. The variable is exported read-only, however, because clients could otherwise destroy the correctness of the module. The implementation of *PriorityQueue* takes the following form:

Priority queue as an abstract data structure

```
MODULE PriorityQueue;
CONST length = 128;
VAR
 n-: LONGINT; (*number of elements*)
 a: ARRAY length OF INTEGER;

PROCEDURE Clear*;
BEGIN n := 0, a[0] := MIN(INTEGER)
END Clear;
```

```
PROCEDURE Insert* (x: INTEGER);
 VAR i: INTEGER;
BEGIN
 IF n < length - 1 THEN
     n := n + 1; i := n;
     WHILE x < a[i DIV 2] DO
         a[i] := a[i DIV 2]; i := i DIV 2
     END;
     a[i] := x
 END
END Insert;

PROCEDURE Remove* (VAR x: INTEGER);
 VAR y, i, j: INTEGER; ready: BOOLEAN;
BEGIN
 IF n > 0 THEN
     x := a[1]; y := a[n];
     n := n - 1; i := 1; ready := FALSE;
     WHILE (i <= n DIV 2) & ~ ready DO
         j := i + i;
         IF (j < n) & (a[j] > a[j+1]) THEN j := j + 1 END;
         IF y > a[j] THEN a[i] := a[j]; i := j ELSE ready := TRUE END
     END;
     a[i] := y
 END
END Remove;

BEGIN Clear
END PriorityQueue.
```

The implementation of the data and the access algorithms is now hidden. Clients see *PriorityQueue* as a black box that is easy to use via its procedures *Clear, Insert* and *Remove.*

Advantages

This solution has several advantages:

(1) Clients do not need to be familiar with the implementation of *PriorityQueue,* which makes it easier for them to use the data structure.
(2) The implementation can be changed later without needing to adapt the clients. If *a* is implemented as a tree rather than as an array (Fig. 3.6), the clients do not notice anything as long as the interface of *PriorityQueue* remains unchanged.
(3) The data are encapsulated in the module *PriorityQueue* and protected there against inadvertent destruction.

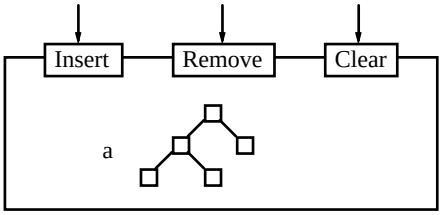

Fig. 3.6 Priority queue with modified implementation but unchanged interface

Drawbacks

Data abstraction also has some disadvantages:

(1) Using *PriorityQueue* is less efficient than using a concrete data structure because access to the data is now channelled through procedures. However, the cost of a procedure invocation is usually low in relation to the cost of the access algorithm.
(2) The data can only be accessed by the operations specified in the interface. If we later need to search for a particular element in the priority queue, say, this would be impossible because the module lacks an appropriate access procedure.

Information hiding

Information hiding should always be used with care and never for its own sake. If all data are hidden as a matter of principle, the simplicity, the flexibility and the extensibility of a module may suffer. One should always be aware of the actual goal: to make the use of a module as easy as possible and to hide changes in its implementation from clients. The module *PriorityQueue* would not have been simplified if *n* had been exported as an access procedure rather than as a variable. The point is not that clients *must not* access private data, but that they *need not* do so in order to use the module.

3.3 Abstract Data Types

Of an abstract data structure there is only one instance. If we need multiple instances, we must use abstract data *types*. An abstract data type is likewise a unit consisting of data and procedures, but contrary to an abstract data structure, it can be used as a type; i.e., multiple variables of this type can be declared.

In Oberon-2 an abstract data type is implemented as a record whose fields can individually be hidden by not exporting them. The priority queue in our example can be implemented as an abstract data type as follows:

```
DEFINITION PriorityQueues;
 TYPE
    Queue = RECORD
       n-: INTEGER  (*number of elements*)
    END;
 PROCEDURE Insert (VAR q: Queue; x: INTEGER);
 PROCEDURE Remove (VAR q: Queue; VAR x: INTEGER);
 PROCEDURE Clear (VAR q: Queue);
END PriorityQueues.
```

Queue is a record whose fields represent the data of the priority queue. Among these fields, *n* is exported read-only, while other fields are hidden (not exported). Note that each variable of type *Queue* has its own set of data.

The access procedures have an additional parameter *q* of type *Queue* designating the record to which the procedures refer. Because the data of the priority queue are changed by the procedures, *q* must be a variable parameter. The implementation of *PriorityQueues* looks like this:

Priority queue as an abstract data type

```
MODULE PriorityQueues;
CONST length = 128;
TYPE
 Queue* = RECORD
    n-: LONGINT; (*number of elements*)
    a: ARRAY length OF INTEGER
 END;

PROCEDURE Clear* (VAR q: Queue);
BEGIN q.n := 0, q.a[0] := MIN(INTEGER)
END Clear;

PROCEDURE Insert* (VAR q: Queue; x: INTEGER);
 VAR i: INTEGER;
BEGIN
 IF q.n < length - 1 THEN
    q.n := q.n + 1; i := q.n;
    WHILE x < q.a[i DIV 2] DO q.a[i] := q.a[i DIV 2]; i := i DIV 2 END;
    q.a[i] := x
 END
END Insert;
```

```
PROCEDURE Remove* (VAR q: Queue; VAR x: INTEGER);
 VAR y, i, j: INTEGER; ready: BOOLEAN;
BEGIN
 IF q.n > 0 THEN
    x := q.a[1]; y := q.a[n]; q.n := q.n - 1; i := 1; ready := FALSE;
    WHILE (i <= q.n DIV 2) & ~ ready DO
       j := i + i;
       IF (j < q.n) & (q.a[j] > q.a[j+1] THEN j := j + 1 END;
       IF y > q.a[j] THEN q.a[i] := q.a[j]; i := j ELSE ready := TRUE END
    END;
    q.a[i] := y
 END
END Remove;

END PriorityQueues.
```

Clients can now create multiple *Queue* variables, e.g.:

```
VAR negNumbers, posNumbers: PriorityQueues.Queue;
```

and use them separately:

```
PriorityQueues.Clear(negNumbers); PriorityQueues.Clear(posNumbers);
…
IF x < 0 THEN PriorityQueues.Insert(negNumbers, x)
ELSE PriorityQueues.Insert(posNumbers, x)
END
```

Extending the language by a new data type

The abstract data type *Queue* can be used like any concrete data type (e.g., INTEGER). The language has been extended by a new data type and thus made better suited to solving a particular problem.

However, abstract data types are again slightly less efficient than abstract data structures, because for each operation the object to which the operation refers has to be passed as a parameter. One should thus give consideration to when an abstract data *type* (i.e., multiple variables of this type) is needed and when an abstract data *structure* suffices. Examples of abstract data types include *Stack*, *Queue*, *Set*, *File*, *Window* and *Text*. On the other hand, for *Mouse* and *Terminal*, abstract data structures suffice because there is normally only one instance of them.

Abstract data types are often pointers

Abstract data types are often implemented not as records, but as pointers to records. Here, too, individual fields of a record can be hidden. The interface of the priority queue then takes the following form:

```
DEFINITION PriorityQueues1;
 TYPE
    Queue = POINTER TO QueueDesc;
    QueueDesc = RECORD
       n-: INTEGER;  (*number of elements*)
    END;
 PROCEDURE Insert (q: Queue; x: INTEGER);
 PROCEDURE Remove (q: Queue; VAR x: INTEGER);
 PROCEDURE Clear (q: Queue);
END PriorityQueues1.
```

The parameter *q* can be a value parameter here because it is not the pointer that is modified by the procedures, but only the fields of the record referenced by the pointer.

4 Classes

A problem with the notation for abstract data types is that data and procedures do not form a syntactic entity. Procedures are declared outside the record and without visible connection to it. Thus it is not immediately clear which procedures belong to a data type.

Therefore, Oberon-2 permits the declaration of special procedures (*methods*) that are syntactically connected to a record. Records that contain methods in addition to data fields are called *classes*. Values whose type is a class are termed *objects*.

Classes differ from abstract data types in that they are extensible and support the dynamic binding of messages to methods. We postpone the discussion of extensibility and dynamic binding to Chapters 5 and 6, respectively.

4.1 Methods

Methods

The procedures associated with a class are termed *methods* or *type-bound procedures* in order to distinguish them from ordinary procedures.

The type *Queue* in Section 3.3, for example, could be implemented as a class with the following interface:

```
DEFINITION PriorityQueues;
TYPE
 Queue = RECORD
    n-: LONGINT;
    PROCEDURE (VAR q: Queue) Insert (x: INTEGER);
    PROCEDURE (VAR q: Queue) Remove (VAR x: INTEGER);
    PROCEDURE (VAR q: Queue) Clear;
 END;
END PriorityQueues.
```

Methods are considered (constant) record fields whose type is a procedure type. At invocation they are accessed like record fields, e.g.:

```
q.Insert(x)
```

Messages

We say that we send the message *Insert* to the object designated by *q*. The terminology should make clear that this is not a procedure call, but a request to an object. Only at run time will it be decided which method is to handle the request.

Receiver

The object to which a message is sent is called the *receiver*. Thus the object designated by *q* is the receiver of the message *Insert*. It reacts by invoking the *Insert* method of its class. Since the variable *q* can contain objects of various classes (see Chapter 5) the *Insert* message can lead to the invocation of different methods.

The receiver is a parameter of every method. In order to distinguish it from other parameters, it is declared in front of the method name:

```
PROCEDURE (VAR q: Queue) Insert (x: INTEGER);
```

Separating the receiver from the other formal parameter seems justified since the corresponding actual receiver parameter is also written in front of the message name when the message is sent:

```
q.Insert(x)
```

Note that the receiver plays a double role: Firstly, it is passed as a parameter to the method, and secondly, the object stored in it determines which method is invoked at run time (see Chapter 6).

Implementation of methods

Let us now look at the implementation of methods. Although they belong to records, it would be unwise to implement them directly in the record declaration. Statements would be in the midst of declarations. Thus in Oberon-2 methods are implemented outside records, but in the same module. Nevertheless, they are considered local to their record. To which record a method belongs can be seen from the type of its formal receiver parameter.

Oberon-2 goes even further and omits the procedure headings in the record declaration. The class interface at the beginning of this section is not an Oberon-2 program, but a piece of documentation created by the browser (see Section 2.6). The actual implementation of *PriorityQueues* takes the following form:

```
MODULE PriorityQueues;
CONST length = 128;
```

```
TYPE
 Queue* = RECORD
    n-: LONGINT; (*number of elements*)
    a: ARRAY length OF INTEGER
 END;

PROCEDURE (VAR q: Queue) Clear*;
BEGIN q.n := 0, q.a[0] := MIN(INTEGER)
END Clear;

PROCEDURE (VAR q: Queue) Insert* (x: INTEGER);
 VAR i: INTEGER;
BEGIN
 IF q.n < length - 1 THEN
    q.n := q.n + 1; i := q.n;
    WHILE x < q.a[i DIV 2] DO q.a[i] := q.a[i DIV 2]; i := i DIV 2 END;
    q.a[i] := x
 END
END Insert;

PROCEDURE (VAR q: Queue) Remove* (VAR x: INTEGER);
 VAR y, i, j: INTEGER; ready: BOOLEAN;
BEGIN
 IF q.n > 0 THEN
    x := q.a[1]; y := q.a[n];
    q.n := q.n - 1; i := 1; ready := FALSE;
    WHILE (i <= q.n DIV 2) & ~ ready DO
       j := i + i;
       IF (j < q.n) & (q.a[j] > q.a[j+1] THEN j := j + 1 END;
       IF y > q.a[j] THEN q.a[i] := q.a[j]; i := j ELSE ready := TRUE END
    END;
    q.a[i] := y
 END
END Remove;

END PriorityQueues.
```

The receiver parameters in the procedure headings of *Clear*, *Insert* and *Remove* indicate that these are not ordinary procedures, but methods of the class *Queue*.

Methods and procedure variables

Why do we actually need a special method notation, since the operations of a class could also be implemented as procedure variables? For example:

```
TYPE
 Queue = RECORD
    n-: INTEGER;
    a: ARRAY length OF INTEGER;
    Insert: PROCEDURE (VAR q: Queue; x: INTEGER);
    Remove: PROCEDURE (VAR q: Queue; VAR x: INTEGER);
    Clear: PROCEDURE (VAR q: Queue);
 END;
```

This is a possible solution, but it has the following drawbacks:

(1) Procedure variables occupy storage in every object, although their values are the same for all objects of a class. Methods, on the other hand, belong to the *class* and are not stored in objects.
(2) Procedure variables must be initialized in each object; this means that they must be assigned procedures whenever an object is created. This is easy to forget. Methods need not be initialized.
(3) The operations of a class should be procedure *constants* rather than procedure *variables*. It should not be possible to exchange them at run time. Methods are constants while procedure variables are not.

Pointer types

Many object-oriented programs do not work with records, but with pointers to records. These are actually pointer-to-class types. For the sake of simplicity, we also refer to these pointer types as classes as long as this does not lead to confusion. Variables of these types point to objects. In the following example *Queue1* is declared as a pointer type:

```
DEFINITION PriorityQueues1;
TYPE
 Queue1 = POINTER TO QueueDesc;
 Queue1Desc = RECORD
    n-: LONGINT;
    PROCEDURE (q: Queue1) Insert (x: INTEGER);
    PROCEDURE (q: Queue1) Remove (VAR x: INTEGER);
    PROCEDURE (q: Queue1) Clear;
 END;
END PriorityQueues1.
```

If the type of the formal receiver parameter is a pointer type, the receiver must be a value parameter, whereas in the case of records it must be a variable parameter. The use of *Queue1* is analogous to *Queue*:

```
VAR q: Queue1;
…
NEW(q); … q.Insert(x); …
```

Comments on the notation for methods

Oberon-2 differs from most object-oriented languages in its notation for methods. Other languages pass the receiver as a hidden parameter with the predefined names *self* or *this*. Oberon-2 avoids hidden mechanisms and requires that the receiver be

explicitly declared as a parameter. The declaration of the receiver also has the advantage that it can be given an expressive name. A name like *q* or *queue* provides better readability than *self*.

When the fields and methods of a receiver are referenced in Oberon-2, they must be qualified with the name of the receiver (e.g., *q.a*). Most object-oriented languages allow the programmer the option of referencing a field as *self.a* or only as *a*. This may be confusing since *a* could be a local or global variable as well.

Omitting the method headings in the record declaration avoids redundancy. The type declaration is kept short. Modifications cannot lead to inconsistent method headings. The browser permits viewing the class with all its methods by extracting this information from the program. This proves faster than flipping through pages of source code.

The fact that the class and module interfaces are not manually written by the programmer, but extracted from the source code, requires some readjustment, especially on the part of Modula-2 programmers. After getting used to the idea, however, anything else seems inconvenient. Programs increasingly tend to be read and written directly on the screen, which makes it practical to enjoy the screen's advantages over paper. Of course, the extracted information can also be printed to hardcopy.

4.2 Classes and Modules

Classes and modules bear certain similarities: they encapsulate data and make them available via access procedures. Are both constructs necessary, or could we scrap modules and employ classes as compilation units?

The question is justified, and some languages, such as Smalltalk, actually use only classes and not modules. Closer examination, however, reveals that using both constructs does make sense. They are complementary.

Information hiding

Classes are expected to support information hiding. In Oberon-2, however, classes are records; access to their fields is unrestricted. How does this agree?

In Oberon-2, not a class but the module in which the class is implemented is responsible for information hiding. Within a module all fields of private classes are visible, but other modules see only the exported fields. Within *PriorityQueues*, field *a* of class *Queue* is visible, but it is not visible for client modules. This makes

sense because a module should only contain related data and procedures anyway. Why should we want to hide information among them?

For reasons of efficiency, it is sometimes necessary for a procedure to have *direct* access to the data of *two or more* classes. If the data were not visible outside the classes, an ordinary procedure could not access them. It would not help to make the procedure a method of one class, for then it still would not have access to the data of the other class. In Oberon-2 the procedure along with the classes to which it must have efficient access can be wrapped in one module. This allows the procedure to access the data of both classes while still keeping the data hidden from other modules. Thus modules permit the grouping of several classes and procedures to a subsystem.

Modules as collections of functions

Not all programs can be forced into the scheme of classes and methods. There are procedures (e.g., numeric functions) that are neither dependent on any state nor modify a state and thus cannot be naturally associated with any class. Modules make it possible to group such functions together, without having to resort to classes which would be an artificial imposition.

Global variables and procedures

Modules allow the use of global variables and procedures in connection with classes. Values that must be accessible for all objects of a class can be stored in global variables of a module without requiring storage in each object. Global procedures permit the execution of operations on a *class*; for example, a procedure could be used to create a new object of a class. Such operations cannot be implemented as methods because an object cannot be sent a message before it is created.

4.3 Examples

The following examples are intended to give the reader a better feel for working with classes.

The class Set

The standard type SET provides sets of integers between 0 and MAX(SET). If sets of arbitrary integers are needed, a class *Set* can be defined:

```
DEFINITION Sets;
TYPE
 Set = RECORD
    PROCEDURE (VAR s: Set) Init (max: INTEGER);
    PROCEDURE (VAR s: Set) CopyTo (VAR s1: Set);
```

```
    PROCEDURE (VAR s: Set) Clear;
    PROCEDURE (VAR s: Set) Incl (x: INTEGER);
    PROCEDURE (VAR s: Set) Excl (x: INTEGER);
    PROCEDURE (VAR s: Set) Contains (x: INTEGER): BOOLEAN;
    PROCEDURE (VAR s: Set) Add (s1: Set);
    PROCEDURE (VAR s: Set) Subtract (s1: Set);
    PROCEDURE (VAR s: Set) Intersect (s1: Set);
 END;
END Sets.
```

Note that *Set* is a record type, and thus the receiver parameter of the methods must be a variable parameter. The meaning of the operations is obvious, so that we can immediately go on to their implementation.

```
MODULE Sets;
CONST setSize = 32;  (*size of type SET*)
TYPE
 Set* = RECORD
    max-: INTEGER;  (*largest element allowed*)
    val: POINTER TO ARRAY OF SET
 END;

PROCEDURE (VAR s: Set) Init* (max: INTEGER);
BEGIN
 s.max := max;
 NEW(s.val, (max + setSize) DIV setSize)
END Init;

PROCEDURE (VAR s: Set) CopyTo* (VAR s1: Set);
 VAR i: INTEGER;
BEGIN
 s1.Init(s.max);
 FOR i := 0 TO s.max DIV setSize DO s1.val[i] := s.val[i] END
END CopyTo;

PROCEDURE (VAR s: Set) Clear*;
 VAR i: INTEGER;
BEGIN
 FOR i := 0 TO s.max DIV setSize DO s.val[i] := {} END
END Clear;

PROCEDURE (VAR s: Set) Incl* (x: INTEGER);
BEGIN
 IF (x > 0) & (x <= s.max) THEN
    INCL(s.val[x DIV setSize], x MOD setSize)
 END
END Incl;

PROCEDURE (VAR s: Set) Excl* (x: INTEGER);
BEGIN
 IF (x > 0) & (x <= s.max) THEN
    EXCL(s.val[x DIV setSize], x MOD setSize)
 END
```

```
END Excl;
PROCEDURE (VAR s: Set) Contains* (x: INTEGER): BOOLEAN;
BEGIN
 RETURN (x > 0) & (x <= s.max)
 & (x MOD setSize IN s.val[x DIV setSize])
END Contains;

PROCEDURE (VAR s: Set) Add* (s1: Set);
 VAR i, max: INTEGER;
BEGIN
 max := s.max; IF s1.max < max THEN max := s1.max END;
 FOR i := 0 TO max DIV setSize DO
     s.val[i] := s.val[i] + s1.val[i] END
END Add;

PROCEDURE (VAR s: Set) Subtract* (s1: Set);
 VAR i, max: INTEGER;
BEGIN
 max := s.max; IF s1.max < max THEN max := s1.max END;
 FOR i := 0 TO max DIV setSize DO
     s.val[i] := s.val[i] - s1.val[i] END
END Subtract;

PROCEDURE (VAR s: Set) Intersect* (s1: Set);
 VAR i, max: INTEGER;
BEGIN
 max := s.max; IF s1.max < max THEN max := s1.max END;
 FOR i := 0 TO max DIV setSize DO
     s.val[i] := s.val[i] * s1.val[i] END
END Intersect;

END Sets.
```

The field *val* of the class *Set* is not exported. Clients can modify it only by means of methods. *val* contains the actual sets of numbers; it is implemented as a dynamic array of sets that is allocated the necessary storage at run time. *max* is the largest element that can be stored in a *Set* object.

Class Figure

As a second example, let us consider a class for figures in a graphics editor. Here we only describe the interface (the module *OS*, which is used in the interface, is described in Appendix B):

```
TYPE
 Figure = POINTER TO FigureDesc;
 FigureDesc = RECORD
     selected: BOOLEAN;
     next: Figure;
     PROCEDURE (Q: Figure) Draw;
     PROCEDURE (Q: Figure) Move (dx, dy: INTEGER);
     PROCEDURE (Q: Figure) Select (x, y, w, h: INTEGER);
     PROCEDURE (Q: Figure) Deselect;
     PROCEDURE (Q: Figure) Load (VAR r: OS.Rider);
     PROCEDURE (Q: Figure) Store (VAR r: OS.Rider);
```

```
END;
```

The class *Figure* is implemented as a pointer to a record. Thus the formal receiver parameters of the methods must be value parameters.

4.4 Common Questions

This section answers some questions that might have arisen in reading Chapter 4.

Q: Can a method and a procedure declared in the same module share the same name?

A: Yes. A method is local to the class to which it belongs. There is no name conflict with globally declared names or with names in other classes.

Q: Can a method be bound to a class that is declared in another module?

A: No. The locality of code and data is an important principle that makes maintenance of software easier. If the methods of a class were distributed among various modules, this would violate the principle of locality.

Q: Can a message be sent to a pointer object if the formal receiver parameter of the method is a record? I.e.:

```
TYPE
   Ptr = POINTER TO Rec;
   Rec = RECORD … END;
VAR
   p: Ptr;

PROCEDURE (VAR r: Rec) M; … END M;

… p.M …  (*is this message legal?*)
```

A: Yes. The record referenced by *p* is passed as a variable parameter to *M*. On the other hand, a message must not be sent to a record object if the formal receiver parameter is a pointer. This means that the following situation is forbidden:

```
VAR r: Rec;

PROCEDURE (p: Ptr) M1; … END M1;
```

```
... r.M1 ...  (*this is illegal*)
```

A record cannot be passed to a pointer. When both variables of type *Ptr* and variables of type *Rec* are used and messages are to be sent to both, the formal receiver parameter of the methods must be declared as a record.