OBJECT-ORIENTED PROGRAMMING WITH SIMULA

Bjørn Kirkerud

OBJECT-ORIENTED PROGRAMMING WITH SIMULA

Bjørn Kirkerud

Institute of Informatics
University of Oslo

ADDISON-WESLEY
PUBLISHING
COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
New York · Don Mills, Ontario · Amsterdam · Bonn
Sydney · Singapore · Tokyo · Madrid · San Juan

Cover designed by Crayon Design of Henley-on-Thames and
printed by The Riverside Printing Co. (Reading) Ltd.
Typeset by the AMS from the author's TEX files.
Printed in Great Britain by Mackays of Chatham plc, Chatham, Kent.

First printed 1989.

British Library Cataloguing in Publication Data
Kirkerud, Bjørn
Object-orientated programming with
Simula.
1. Computer systems. Programming
languages: Simula language
I. Title
005.13'3

ISBN 0–201–17574–6

Library of Congress Cataloging in Publication Data
Kirkerud, Bjørn
Object-oriented programming with SIMULA/Bjørn Kirkerud.
p. cm. – (International computer science series)
Bibliography: p.
Includes index.
ISBN 0–201–17574–6
1. SIMULA (Computer program language) 2. Object-oriented programming. I. Title. II. Series.
QA76.73.S55K57 1989
005.13'3–dc19

89–149
CIP

Preface

Overview

A computer program is constructed in order to get a computer to behave in a desired manner. The aim of this book is to teach the reader to construct computer programs in a systematic and efficient fashion.

There are many different approaches to program construction. Some people feel the differences to be so substantial that it is justified to talk about different paradigms for programming. One approach that has received much acclaim during the last few years is called *object-oriented programming*. After more than 25 years of experience in programming and teaching programming, and after having used various approaches, I feel the praise given to the object-oriented approach to be justified, and have chosen to base this book upon it.

A programming language is a notational system for writing programs. A programming language called Simula has been chosen for use in this book. The reason for this choice is not that Simula is a very well known or broadly used programming language, but rather that it is well suited as a vehicle for learning programming in general and object-oriented programming in particular. The main ideas of object-oriented programming were in fact introduced for the first time in Simula as early as 1967. Many people who have learned programming via Simula have experienced not only that it is quite easy to adjust to other programming languages but that they quickly become better users of those other languages than many people who have used those languages for a long time. Such ability to adjust to new tools (a programming language is of

course a tool) – which certainly is of particular value in an area which changes rapidly and in which new tools incessantly are introduced – seems to be less commonly experienced by people who have had other languages as their first programming language.

It is important to recognize the distinction between *learning a programming language* and *learning how to construct programs*. The latter is the aim of this book, and I have not aimed at giving a complete coverage of every part of Simula. Nevertheless, most features of Simula are described in the book. The only major omission is that part of Simula which deals with so-called discrete event simulation. In addition, I have omitted some details which I think are obsolete or of minor general interest.

Readership

No previous knowledge of computers and computer programming should be necessary to read this book: I have assumed that the reader knows nothing about computers and have explained everything from scratch. In particular I have not assumed that the reader has constructed any computer programs or knows any programming language. But the first few chapters are rather tersely written, and some experience in using and/or programming computers (for instance a simple course in a programming language like Pascal or Logo) would certainly help the reader when he or she reads those chapters.

A reader who is well acquainted with traditional programming and who knows the programming language Pascal (or something similar) should be able to read most of the first seven chapters rather quickly. But he or she is advised not to skip these chapters, because they contain many details that will be new.

Background and related material

The choice of material and examples, and the sequence in which different concepts are introduced, are based upon experience gained from various courses in programming I have given (either alone or in collaboration with others) since 1962, mostly at the University of Oslo. Some parts of the book are translated from my book *Systematisk Programmering* (in Norwegian).

Every program in the book has been tested with care on one or more computers. Nevertheless, they may contain errors, and I cannot guarantee that they will always behave as indicated.

If you have access to a computer that is connected to the computer network called 'Internet', you may obtain copies of the programs in the book and also proposed solutions to some of the exercises. Every program (except some of the smaller ones) may be retrieved by 'anonymous ftp' from directory simbook on host ifi.uio.no.

Acknowledgements

I would like to thank my colleagues Ole-Johan Dahl, Stein Gjessing, Stein Krogdahl and Olaf Owe for valuable comments, inspiring criticisms and the discovery of errors in preliminary versions of some of the chapters in this book. A number of students, one of which is Kjetil Otter Olsen, have also given helpful comments on parts of the manuscript.

The book was written on a Sun computer running under an operating system called UNIX. I have used a text processing system called Gnu-emacs to enter the text, and the document preparation system LaTeX to typeset the manuscript and to 'draw' the figures. I am not an expert at setting up and tuning systems like these to behave as desired, and am grateful for the assistance of several people, in particular Jens Thomassen and my son Ketil Kirkerud. Finally, I would like to thank Kjell E. Nordli who used the Musikus-system (which has been developed at the University of Oslo) to produce the musical score in Figure 1.1.

Bjørn Kirkerud
Oslo, September 1989

Contents

Preface		v
Chapter 1	**Learning how to Construct Programs**	1
1.1	Programming languages	3
1.2	Syntax, semantics and pragmatics	4
1.3	Programs may have side effects	6
1.4	About the necessity for practice	7
Chapter 2	**A Simple Program**	9
2.1	Breakdown of a simple program	11
2.2	How to get the program executed by a computer	17
	Exercises	19
Chapter 3	**A Small Part of Simula**	21
3.1	Words and symbols	21
3.2	Syntax diagrams	26
3.3	Declarations	28
3.4	Expressions	32
3.5	Imperatives	41
3.6	Programs	53
3.7	A common error	54
3.8	Comments	55
3.9	Layout	57
	Exercises	61
Chapter 4	**Arrays and Blocks**	67
4.1	A first sketch of a program	68
4.2	A more detailed program sketch	69
4.3	How to declare a table of variables	70
4.4	Reading a list of numbers	71

4.5	The histogram	76
4.6	Step imperatives	77
4.7	A complete program according to proposal 1	80
4.8	Proposal 2: ideas and sketches	81
4.9	Blocks	82
4.10	A complete program according to proposal 2	84
4.11	Robustness	86
4.12	More about arrays	88
4.13	More about step imperatives	91
4.14	Some frequently used program patterns	92
	Exercises	96
Chapter 5	**Simple Use of Files**	**101**
5.1	Tools and tool-boxes	102
5.2	Simple tools for writing to files	102
5.3	Simple tools for reading from files	108
5.4	Commonly used program patterns	112
5.5	Sysin and sysout	117
5.6	An example: counting words in a file	118
	Exercises	120
Chapter 6	**Procedures**	**125**
6.1	Procedures with no parameters	126
6.2	Procedures with parameters	131
6.3	Why use procedures?	144
6.4	External procedures	153
	Exercises	154
Chapter 7	**Case 1: A Program for Weather Data**	**163**
7.1	Specifications for a program	163
7.2	A first rough draft	164
7.3	How to give and take commands	165
7.4	The data structure	168
7.5	A first test version	168
7.6	Programming the procedures	171
7.7	A program that works	174
	Exercises	181
Chapter 8	**Case 2: A Program for Students and Classes**	**183**
8.1	Rough specifications for the program	184
8.2	A rough draft and a first test program	185
8.3	Two alternative data structures	188
8.4	Objects	189
8.5	Simula classes	190

8.6	A second test program	195
8.7	More constructions for handling objects	198
8.8	How to write a student's data	200
8.9	A simple program	201
8.10	Objects with data and operations	204
8.11	Protection and encapsulation	205
8.12	Drafts for the command procedures	206
8.13	Demands on the data structure	208
8.14	Programs for the command procedures	212
8.15	A third test program	213
8.16	Data structure for 1500 students	215
8.17	A program that satisfies the specifications	221
8.18	Prefixed blocks	223
8.19	Sorting students	224

Chapter 9	**Objects and Classes**	**229**
9.1	Declaration of simple classes	229
9.2	How to generate objects	231
9.3	Reference types	232
9.4	Remote access of attributes	236
9.5	Inspecting objects	238
9.6	Classes with parameters	240
9.7	Classes with imperatives	242
9.8	References as parameters and function values	243
9.9	Comparing references	245
9.10	How to copy objects	246
9.11	Subclasses and inheritance	247
9.12	Virtual procedures	261
9.13	How to stop and restart objects	265
9.14	How to protect and hide attributes	268
9.15	Prefixed blocks	269
9.16	External classes	269
	Exercises	271

Chapter 10	**Case 3: Can a Computer Learn?**	**277**
10.1	'Learning' to play tic-tac-toe	277
10.2	The game of Nim	278
10.3	Nim-players	279
10.4	Games and tournaments	281
10.5	Human Nim-players	283
10.6	Nim-playing machines	284
10.7	Perfect Nim-players	285
10.8	Machines that 'learn' to play	286
10.9	Training against a 'sparring partner'	291
10.10	Can computers 'learn' real games?	293
	Exercises	294

Chapter 11	**Texts**	**297**
11.1	Author profiles	297
11.2	Syntax and semantics of texts	315
	Exercises	327
Chapter 12	**Files**	**331**
12,1	Files in general	331
12.2	A hierarchy of file-packages	336
12.3	Outfile	339
12.4	Infile	346
12.5	Placing texts in images	351
12.6	Printfile	355
12.7	Sysin and sysout	356
12.8	Directfiles	362
12.9	Another weather data program	364
12.10	Bytefiles	375
12.11	Access modes	377
	Exercises	379
Chapter 13	**Case 4: An Interpreter**	**381**
13.1	Syntax and semantics of Simpla	381
13.2	Drafts for the interpreter	384
13.3	Imperatives	387
13.4	Sequences of imperatives	397
13.5	Integer expressions	400
13.6	Boolean expressions	402
13.7	Operators and comparators	403
13.8	States	404
13.9	Various auxiliary procedures	406
13.10	An interpreter for Simpla	407
13.11	Getting the interpreter to work	408
	Exercises	412
Chapter 14	**Tables, Chains and Sets**	**415**
14.1	Some common operations on lists	415
14.2	Tables of pointers	416
14.3	Pointer chains	421
14.4	How to choose a data structure	432
14.5	Sets of objects	433
14.6	Sequences of objects	438
14.7	An example: data on students and courses	441
14.8	Simulating a post office	446
	Exercises	456

Appendix A Predefined Identifiers 463

A.1 Basic procedures 463
A.2 Time and date 464
A.3 Text procedures 464
A.4 Implementation-dependent procedures 465
A.5 Mathematical functions 466
A.6 Random numbers 466

Appendix B The ISO Characters 469

Appendix C Syntax Diagrams 471

C.1 Program 471
C.2 External specifications 471
C.3 Imperatives 473
C.4 Declarations 476
C.5 Expressions 480
C.6 Names and parameters 484
C.7 Literals 485
C.8 Identifiers 486

Appendix D Tools for Sequences and Sets 489

D.1 Main structure of the class settools 489
D.2 The class element 489
D.3 The class sequence 490
D.4 The class basis_set 491
D.5 The class set 492
D.6 The class ordered_set 493
D.7 The class table 494

References 497

Index 499

Chapter 1
Learning how to Construct Programs

According to my dictionary (which predates the invention of computers and computer programs), **a program is a prearranged plan or course of proceedings**. Most people are familiar with programs of various kinds. Some examples:

- A musical score can be looked upon as a very detailed program for a musician or an orchestra. Figure 1.1 contains the first parts of one the most beautiful programs ever written.
- A knitting pattern is a program for a knitter.
- A recipe for bread-baking is a program for a cook.
- A **computer program** is a detailed plan for a computer. Figure 2.1 (p. 12) in the next chapter contains the first of many examples of such programs in this book.

Observe that programs are intended to be **executed** by someone or something: musicians, cooks, knitters or computers. Programs consist of **sequences of instructions** to be obeyed by the program executor(s). These instructions must be written in a **language** – or **notational system** – which must be understood by whatever or whoever executes the programs. But the language used may be quite incomprehensible to anyone not 'in the field'.

A computer program like the one in Figure 2.1 (see p. 12) is probably unintelligible to anyone unfamiliar with the language in which it is written. But it is not difficult to learn enough about computers and about the language in which the program is written to understand its meaning and also the meaning of similar programs – at least if they are not overly complex. A reader who takes time to study the next chapter should be able to understand not only the program explained there but quite a lot of other programs. An experience shared by many is that it is rather easy to learn to *read* programs which are

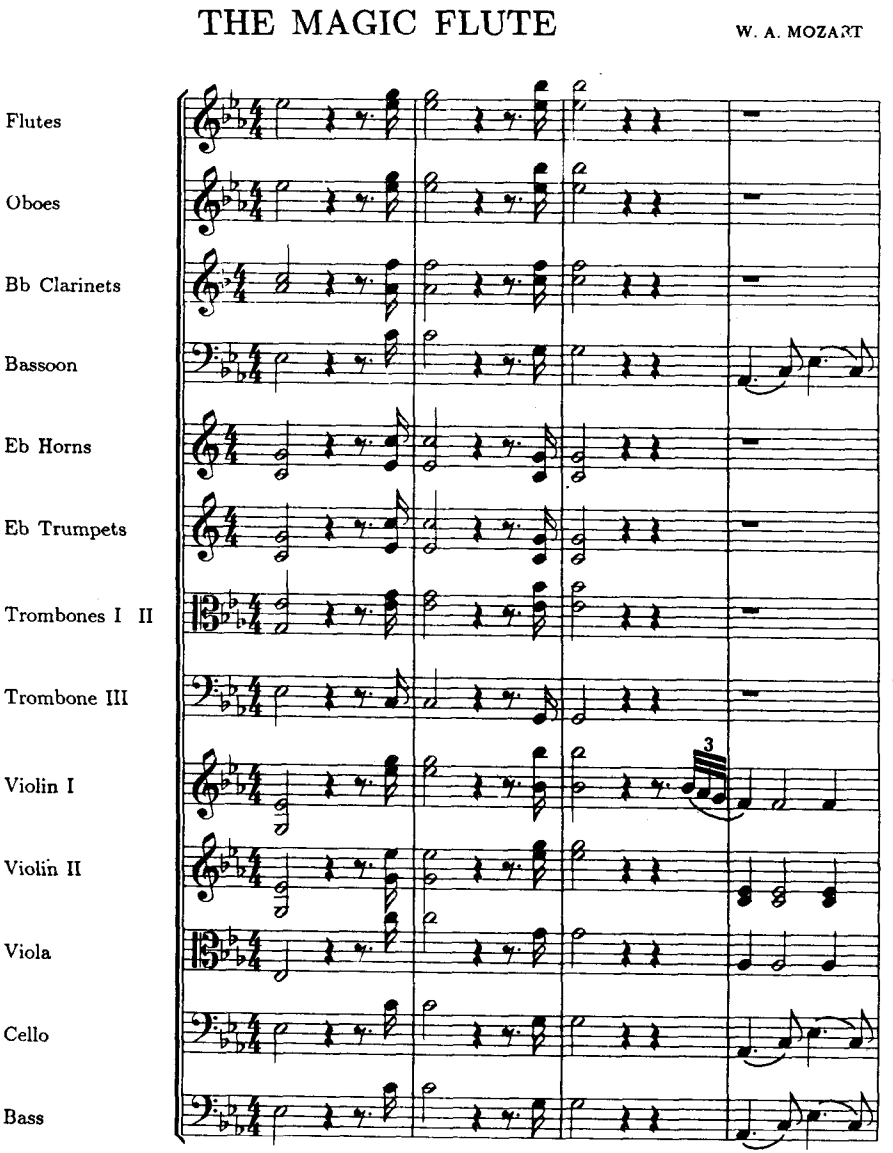

Figure 1.1 A program for an orchestra.

not too complex and to understand what happens when they are executed by a computer. But to learn enough to be able to *construct* programs is a very different matter. The primary aim of this book is to teach the reader how to construct programs. But first, some general observations and advice concerning programs and program construction.

1.1 Programming languages

A programming language is a notational system for writing computer programs: that is, for writing sequences of instructions intended to be executed by computers. A large number (too large, according to most observers) of such languages have been constructed during the short time man has had the need to write computer programs. Some of the most widely used and well known of these languages are FORTRAN, COBOL, Pascal, C, Ada and LISP. The language we use in this book is called Simula.

It is important to recognize the distinction between *learning a programming language* and *learning how to construct programs*. This distinction has some similarity to the distinction between learning the precise rules for writing musical scores and learning how to compose music: the former is not very difficult (and arguably not very interesting in itself), while the latter is rather more demanding and also more rewarding. My aim is to teach the reader how to construct programs, a more valuable (and also more difficult!) skill than mastering the quirks and foibles of some specific programming language.

To achieve this goal, it is certainly necessary to use a programming language, which the reader must learn. I have chosen Simula for this purpose. One reason for this choice is that, in my opinion, Simula is very well suited for learning how to construct programs in general. It is my experience – based upon many years of teaching thousands of students – that it is quite straightforward to learn to use most other programming languages after having learned the basic principles of programming and programming methodology using Simula as a vehicle. Many students who take what at first seems to be a detour via Simula become better programmers of languages like FORTRAN, C, Ada or COBOL (and other programming languages) than those who go directly to these languages. This may perhaps seem strange: it appears to be akin to being told that the best way for an English-speaking person to learn French is by first learning German, which is patently unsound advice. But this analogy is only superficial. First, programming languages are very different from languages like English and French. It is somewhat misleading to say that Simula and FORTRAN are languages: it would be better to call them notational systems. Compared to ordinary human languages, programming languages are extremely simple in structure, and also very much simpler to learn. The experience many of us have of difficulty in learning foreign languages is therefore not valid when it comes to learning programming languages. Second, it is important to recognize that our aim

is to teach the reader principles and methods that are of general use when constructing programs, and not just a notational system for writing programs. As already stated, *the aim is to teach programming, not just a programming language.* Simula contains a number of very powerful and generally useful concepts which do not exist in most other programming languages and which are of particular interest when it comes to learning generally applicable methods of programming.

1.2 Syntax, semantics and pragmatics

When studying a language, be it one of the ordinary human languages like English, Latin or Esperanto, or a programming language like Simula or COBOL, it can be advantageous – and is in fact quite common – to split the study into three parts: syntax, semantics and pragmatics. These parts are, however, closely related, and it is advisable to study them simultaneously.

1.2.1 Syntax

The **syntax** of a language consists of rules for correct use of the language. When studying syntax, one learns how to use the language without making grammatical errors. In ordinary human languages (such languages are often called **natural** languages in order to distinguish them from the artificial programming languages), there are rules for inflection, declension and conjugation, for correct spelling, for hyphenation, for constructing new words, and for much more. The rules are usually rather complex, with many special cases and exceptions.

The syntactical rules of any programming language are very much simpler than the syntactical rules of any natural language: there are fewer and simpler rules, and these rules are almost without exceptions. But the syntactical rules of a programming language must be adhered to much more strictly than we have to adhere to the syntactical rules of a natural language. If we deviate somewhat from the syntactical rules when we use a natural language like English, people will usually understand us, at least if the deviations are not too gross and misleading. But even the slightest syntactical mistake made when using a programming language will normally entail that the program will not be accepted for execution. Instead, the computer that should execute the program will produce a message saying that there are syntax errors in the program.

Most students of a programming language struggle a little with the syntax, especially in the beginning. But after a while, they find the syntax easy to learn. One reason for this is that they get help from the *compiler.* This is a program that translates from the programming language to the internal language of the computer (which consists only of 0s and 1s, and is quite unreadable for humans). The compiler – which is a very large program –

has already been constructed (by specialists), and is ready for use by Simula programmers. You do not have to construct it. It starts by checking that the program to be translated is without syntactical errors. If the program is without any errors, it is translated and executed. If, however, errors are found, the compiler does not proceed with the translation. Instead, it writes messages that describe the errors. This means that any syntactical errors are caught quite soon after they have been committed, and that the programmer gets almost immediate notification of his or her errors.

1.2.2 Semantics

The **semantics** of a language is concerned with the *meaning* of syntactically correct constructs of the language. The semantics of a natural language is learned partly by reading and hearing definitions and explanations, partly by experience (sometimes frustrating or even painful). The semantics of a programming language describes what effects programs in the language have when they are executed by a computer. It is normally learned in a fashion similar to that in which the semantics of an ordinary language is learned: some is learned by reading and hearing descriptions which explain what will happen when programs in the language are executed by a computer, and some by seeing what happens when programs are executed.

Most parts of a programming language have rather simple semantics, not very difficult to learn. But most programming languages have parts with more intricate semantics, not easily understood at first or even second glance. It is advisable for a beginner to avoid such intricacies, and stay with the more easily grasped parts of the language. When these parts are mastered, it is time to proceed to the more complex parts of the language.

1.2.3 Pragmatics

The **pragmatics** of a language contains rules for proper and good use of the language. This is the part of a programming language that is most difficult to learn, and also the part that this book particularly stresses.

A good program is almost always constructed in order to solve a given problem, and not, for instance, written in order to show off the programmer's mastery of the more subtle intricacies of programming. The most important demand to be satisfied by a program is therefore that it works correctly and solves the problem it was constructed to solve. Rules and methods that may aid a programmer in his or her construction of a program which solves a given problem are also considered part of the pragmatics of the programming language. The importance of such rules and methods grows strongly with the size and complexity of the problems to be solved and the programs to be constructed. A problem when writing a book like this is that the importance of some of these rules and methods only becomes apparent for problems with

a size and complexity beyond what may be presented as one of many examples in a book. Using some of these methods on small problems may sometimes be similar to using cannons to shoot sparrows.

It is furthermore important to realize that the work on most programs that are used in serious applications – and not just constructed as part of a course in programming – almost never terminates: new wishes and demands for the programs usually turn up all the time. It is also a very common experience to discover errors in programs as they are being used. This means that programs frequently must be changed. It is therefore very important to construct the programs in such a way that they are easy to change. By taking a little care in the original construction of a program, later amendments and corrections may often be done without excessive effort.

1.3 Programs may have side effects

Many programs that are constructed to solve some given problem may have **side effects** when being used. For example, the intended main effect of using a computer with a suitable program to control an industrial process in a factory may be to achieve a more rational and economic production. One side effect may be to reduce the number of employees in the factory (and hence possibly to increase unemployment in the surrounding society); another to dramatically change the character of the work for the remaining workers. Another example is the use of a computer with a suitable program to train children in correct spelling of English words, which may reduce the number of spelling errors committed by the children and possibly also the number of teachers (and hence the size of the school budget), but will in addition certainly affect the more human and social aspects of the education. As a final example, to hand over the control of a weapon or a weapon system (for instance, a rocket) to a computer may increase the speed with which the weapon may be used in a crisis, but will also influence the chances that the weapon is used accidentally owing to errors in the computer or the program or the input to the computer.

It is frequently the case that the various side effects of a program may be of greater importance than the planned main effects. Such side effects are not always good and desired, at least not for all those who are exposed to them. A programmer should not close his or her eyes to the possibility of undesired side effects, but should as much as possible try to foresee them. Like everybody in a society, a programmer is responsible for his or her actions and also for the effects these actions may have on other people or on nature.

Many countries have laws and agreements that set limits for how programs may be constructed and used. For example, in some countries, the employees of enterprises have the right to influence – even participate in – the development of programs which may affect their working conditions. Another example: there may be laws that limit the kind of information that may be