

Information technology — Programming languages —

Part 3: Modula-2, object-oriented Modula-2

ICS 35.060

NO COPYING WITHOUT BSI PERMISSION EXCEPT AS PERMITTED BY COPYRIGHT LAW



Document Sharing Hub

National foreword

This British Standard reproduces verbatim ISO/IEC 10514-3:1998 and implements it as the UK national standard.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and systems software interfaces, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or by using the “Find” facility of the BSI Standards Electronic Catalogue.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, pages i and ii, the ISO/IEC title page, pages ii to iv, pages 1 to 36, an inside back cover and a back cover.

This standard has been updated (see copyright date) and may have had amendments incorporated. This will be indicated in the amendment table on the inside front cover.

This British Standard, having been prepared under the direction of the DISC Board, was published under the authority of the Standards Committee and comes into effect on 15 May 1999

© BSI 03-2000

ISBN 0 580 29337 8

Amendments issued since publication

Amd. No.	Date	Comments

Contents

	Page
National foreword	Inside front cover
Foreword	iv
Text of ISO/IEC 10514-3	1

INTERNATIONAL
STANDARD

ISO/IEC
10514-3

First edition
1998-12-01

**Information technology — Programming
languages —**

**Part 3:
Object Oriented Modula-2**

*Technologies de l'information — Langages de programmation —
Partie 3: Modula 2 orienté objet*



Reference number
ISO/IEC 10514-3:1998(E)

AVA Documents
Document Sharing Hub

Contents

	Page
Foreword	iv
Introduction	1
1 Scope	1
1.1 Goals	1
1.2 Specifications included in this part of ISO/IEC 10514	1
1.3 Relationship to ISO/IEC 10514-1	1
1.4 Specifications not within the scope of this Standard	1
2 Normative references	2
3 Definitions, Structure, and Conventions	2
3.1 Definitions	2
3.2 Structure of the Formal Definition	2
3.3 Conventions	2
4 Requirements for Implementations	2
4.1 Translation	2
4.2 Source Code Representation	2
4.3 Ordering of Declarations	2
4.4 Predefined Entities	2
4.5 Library Modules	2
4.6 Errors	3
4.7 Exceptions	3
4.8 Implementation-dependencies	3
4.9 Documentation	3
4.10 Statement of Compliance	3
4.11 Minimum requirements	3
5 The Lexis	3
5.1 Additional Keywords	3
5.2 Additional Pervasive Identifiers	3
6 Language	4
6.1 Classes	4
6.1.1 Class Definition	5
6.1.2 Class Declaration	6
6.1.3 Abstract Classes	7
6.1.4 Traced Classes	8
6.1.5 Reveal Lists	9
6.1.6 Inherit Clause	9
6.1.7 Class Components	10
6.1.8 Overridden Methods	10
6.1.9 Constructor	11
6.1.10 Destructor	11
6.2 Access to Class Components	11
6.2.1 Access by Class Identifier	12
6.2.2 Access by Object Selection	12
6.2.3 Access in Subclasses	13
6.2.4 Immutable entities	13
6.3 Object Variables	13
6.3.1 Assignment	13
6.3.2 Comparison	14
6.3.3 Creation	14
6.3.4 Destruction	15
6.3.5 Empty Reference	15

	Page
6.3.6 Traced Object References	15
6.4 Membership Test	17
6.5 Guard Statement	17
6.5.1 Guarded Statement Sequence	17
6.6 Safeguarded Modules	18
6.7 Garbage Collection	18
6.8 Changes to the Base Language	19
6.8.1 Module Header	19
6.8.2 Definitions and Declarations	20
6.8.3 Blocks	21
6.8.4 Statement Part	21
6.8.5 Parameter Compatibility and Argument Binding	21
6.8.6 The Module COROUTINES	21
6.8.7 Environment	22
7 Required System Modules	22
7.1 Object Exceptions	22
7.1.1 The Interface to M2OOEXCEPTION	22
7.1.2 The Entities of M2OOEXCEPTION	22
7.1.3 Aggregation of the Exceptions of the Language Extensions	22
7.2 The Module GARBAGECOLLECTION	23
7.2.1 The interface to GARBAGECOLLECTION	23
7.2.2 The Procedures of GARBAGECOLLECTION	23
7.2.3 The Functions of GARBAGECOLLECTION	23
Annex A (informative) Collected Concrete Syntax	24
A.1 Class Definition	24
A.2 Class Declaration	25
A.3 Reveal List	26
A.4 Inherit Clause	26
A.5 Designators	26
A.6 Guard Statement	26
Annex B (informative) Changes to the Syntax of the Base Language	27
Annex C (informative) Glossary	28
Annex D (informative) Examples	30
D.1 Class Definition and Class Declaration	30
D.2 Cross Linked Classes	31
D.3 Inheritance and Overridden Methods	31
D.4 Abstract Class Definition	32
D.5 Creation and Destruction	33
D.6 Constructor and Destructor	33
D.7 Type Inquiries	34
D.8 Constructor chain and Destructor chain	34
D.9 Coroutines and Garbage Collection	34
Annex E (informative) Finalization Order	35
Annex F (informative) Participating Individuals and Organisations	Inside back cover

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 10514-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

ISO/IEC 10514 consists of the following parts, under the general title *Information technology — Programming languages*:

- *Part 1: Modula-2, Base Language*;
- *Part 2: Generics in Modula-2*;
- *Part 3: Object Oriented Modula-2*.

Annex A to Annex F of this part of ISO/IEC 10514 are for information only.

Introduction

This part of ISO/IEC 10514 is part three of the multi-part standard ISO/IEC 10514 and specifies the form and meaning of Object Oriented Modula-2 programs and by reference to that specification lays down requirements for Object Oriented Modula-2 implementations.

The reader is referred to International Standard ISO/IEC 10514-1 (the first part of this multipart standard, herein referred to as “the Base Standard”) for introductory remarks on the programming language Modula-2.

This part of ISO/IEC 10514 defines Object Oriented Modula-2 by additions to the Base Language without changing the meaning of any parts of the Base Language (except for the introduction of new keywords — see clause 5).

This part of ISO/IEC 10514 does not provide a formal specification of Object Oriented Modula-2, although it is the intention of WG13 to construct the appropriate VDM-SL descriptions for the syntax and semantics described herein when committee resources permit.

Rationale

Object oriented programming is a method of programming that allows a high degree of abstraction as well as good structuring of programs. Because of its substantial benefits it has become a common method of programming.

As Modula-2 in its original design provides for basic facilities necessary for object orientation (like data encapsulation and modularization), full object oriented facilities can be easily added to the base language in a very natural way. Thus the advantages of this new programming method are made available to the programmer in a fully upward compatible way.

1 Scope

1.1 Goals

In addition to the goals of the Base Language, the goal of this part of ISO/IEC 10514 is to provide simple extensions to allow object oriented programming facilities to be added to the Base Language defined in International Standard ISO/IEC 10514-1 without altering the meaning of any valid program allowed by the Base Language (except for the use of the new keywords introduced by this standard, see clause 5).

1.2 Specifications included in this part of ISO/IEC 10514

In addition to the specifications included in the Base Language this part of ISO/IEC 10514 provides specifications for:

- required symbols for Object Oriented Modula-2 programs;
- the lexical structure, the syntactic structure and semantics of Object Oriented Modula-2 programs;
- the interface to and the semantics of Object Oriented Modula-2 system modules;
- violations of the rules for the use of the object oriented extensions that a conforming implementation is required to detect;
- further compliance requirements for implementations, including documentation requirements.

1.3 Relationship to ISO/IEC 10514-1

This part of ISO/IEC 10514 is part three of the multi-part standard ISO/IEC 10514. This part of ISO/IEC 10514 extends and modifies the Base Language ISO/IEC 10514-1, but the adoption of this part of ISO/IEC 10514 is optional with respect to the Base Language. This part of ISO/IEC 10514 is also independent of any other parts of ISO/IEC 10514 except for part 1, and can be adopted either together with or independently of such other parts.

1.4 Specifications not within the scope of this part of ISO/IEC 10514

In addition to the categories of specifications excluded by the Base Language this part of ISO/IEC 10514 provides no specifications for:

- the internal representation of the objects and their associated methods;
- the implementation of the garbage collector;
- the implementation of the tracking mechanism for traced objects.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 10514-1:1996, *Information technology — Programming languages — Part 1: Modula-2, Base Language*.

3 Definitions, Structure, and Conventions

3.1 Definitions

For the purposes of this part of ISO/IEC 10514, all the definitions contained in the Base Standard apply. No additional definitions are needed.

NOTE A glossary of the terms used in this part of ISO/IEC 10514 is found in Annex C.

3.2 Structure of the Formal Definition

This part of ISO/IEC 10514 states its requirements in the same form as the Base Language with the exception that it does not include formal expression of semantics in VDM-SL at this time.

3.3 Conventions

The conventions used in this part of ISO/IEC 10514 are to be interpreted in the same way as in the Base Language with the exception that this document does not include VDM-SL at this time.

4 Requirements for Implementations

A conforming Object Oriented Modula-2 implementation shall meet the requirements for Modula-2 implementations that are laid down in the Base Language.

In addition:

4.1 Translation

A conforming Object Oriented Modula-2 implementation shall accept compilation modules for translation from source code when they contain the additional lexical form defined in clause 5, and when this is in the syntactic form specified in clause 6. A conforming Object Oriented Modula-2 implementation shall also accept the lexical forms defined in the Base Language when they are used in the (new) syntactic forms specified in this Object Oriented Modula-2 standard.

4.2 Source Code Representation

A conforming Object Oriented Modula-2 implementation shall provide the additional keywords specified in clause 5 and shall recognize keywords and identifiers as specified in that clause, including those situations where keywords and symbols from the base language are used for new syntactic constructs in this document.

4.3 Ordering of Declarations

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation.

4.4 Predefined Entities

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation.

4.5 Library Modules

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation. No new library modules are specified by this document.

4.6 Errors

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation, with the following changes:

- it shall detect any new errors defined in this document in a manner consistent with the detection and reporting of errors required by the Base Language.
- In standard mode a conforming implementation shall treat the use of extensions that are not specified by this standard or by any other part of this multi-part standard as errors. Conformance to standards parallel to this one (if any) is on an additive basis, so that two or more such standard extensions can be conformed to simultaneously.

NOTE The intent of this provision is to allow a version of Modula-2 to support, for example, both object oriented and generic.

4.7 Exceptions

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation. This includes the additional exceptions for object oriented extensions as identified in subclause 7.1.3. This subclause also specifies which exceptions are mandatory and which exceptions are not mandatory.

4.8 Implementation-dependencies

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation.

4.9 Documentation

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation.

4.10 Statement of Compliance

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation and in addition, a separate compliance statement shall be made citing the degree of compliance with this standard.

4.11 Minimum requirements

A conforming Object Oriented Modula-2 implementation shall have rules identical in this respect to a conforming Modula-2 implementation.

5 The Lexis

The lexis of Object Oriented Modula-2 is based directly on the lexis of the base language as defined in ISO/IEC 10514-1. All syntax elements of the base language keep their representation and meaning; new keywords and pervasive identifiers that are needed to express the new language elements are defined in this clause.

NOTE A conforming program written in the base language is compiled correctly by a translator conforming to this standard unless one or more of the newly defined keywords are used as an identifier in that program.

5.1 Additional Keywords

Standard Object Oriented Modula-2 includes the following keywords in addition to those used in the Base Language:

“AS”		“ABSTRACT”		“CLASS”	
“GUARD”		“INHERIT”		“OVERRIDE”	
“READONLY”		“REVEAL”		“TRACED”	
“UNSAFEGUARDED”					

5.2 Additional Pervasive Identifiers

Standard Object Oriented Modula-2 includes the following identifiers in addition to those used in the Base Language:

“CREATE”		“DESTROY”		“EMPTY”	
“ISMEMBER”		“SELF”			

6 Language

Object Oriented Modula-2 is an extension of the Base Language to provide for facilities for object oriented programming. The provided model has reference-based semantics and single inheritance; no facilities for type parameterization have been included.

Summary of model characteristics

The following list provides an overview of the model.

- Access to objects: via references only.
- Arity of inheritance: single inheritance.
- Visibility modes: three modes via explicit visibility rules — hidden, family (for those developing subclasses) and public.
- Constructors/Destructors (without parameters): class initialization/finalization (similar to module initialization/finalization).
- Object allocation/deallocation: via “CREATE” and “DESTROY” (untraced classes)/implicit (traced classes).
- Type inquiries: test for membership of a class-type family and selection in a guarded region.
- Class syntax: different from modules and records because of new concepts and similar to modules and records because of visibility rules and type nature.
- Garbage collection: on dedicated classes.
- Standard root object: none.
- Genericity: none; this is subject of a separate part of this multi-part standard.
- Operator definition: no.
- Operator overloading: no.
- Method covariance: no.

Reference-based semantic model

The model for object-oriented programming for standard Object Oriented Modula-2 is to use reference-based semantics; i.e. object variables contain references to objects and assignment copies references and not objects themselves. For example, if x and y are object variables, $x := y$ results in x having the same reference to an object that y has; it does not copy to x the contents of the object referenced by y .

NOTE The consequence of this model is to allow objects to be created only dynamically.

6.1 Classes

A new kind of entity is introduced, called a *class*. Classes are a new kind of type (called *class type*) — the type of reference based objects — and can be used as any other type (see 6.8.7). Classes are declared in implementation or program modules (referred to as “class declaration”). In addition to a class declaration a class definition may occur in the corresponding definition module. The class definition specifies an external interface for a class in the same way that a procedure definition specifies an external interface for a procedure.

A class declaration or class definition opens a new scope (details see 6.1.1 and 6.1.2). All identifiers declared in a class definition are also visible in a class inheriting from this class (they are referred to as *family visible*). The identifiers declared in a class declaration are only visible in the class declaration of a subclass in the same module; to subclasses declared in other modules they remain hidden (see 6.1.6). The reveal clause allows one to specify components that may be accessed from outside the class or its descendants; they are further referred to as public components (see 6.1.5).

Abstract classes (and methods) are further described in subclause 6.1.3.

A class may inherit from another class; this is further described in subclause 6.1.6.

The variable (state) components of a class do not denote variables in the usual way, i.e. their definition does not imply the association of storage with them. They act more like fields of a record type declaration.

The procedures declared in a class are called methods of the class. They always act on objects of this (or a descendant) class and have a reference to this object as an implicit first parameter (see 6.1.7). They are not compatible with normal procedure types and there are no method types, method variables or method constants. Methods can be overridden in subclasses as described in subclause 6.1.8.

Finally, classes may contain a constructor and a destructor, this code is to be executed at the time of object creation and deletion (see 6.1.9 and 6.1.10).

As the meaning of traced and untraced classes is different only with respect to objects, no distinction between these two kinds of classes is made in the description of the declaration of classes.

6.1.1 Class Definition

A class definition defines a new structured type. Each value of the type has a collection of components, called components of that class. The components are identified by their names. The components can be accessed from outside the class via class name qualification or via an object designator (see 6.2).

NOTE 1 The term “component” is already used in the Base Language (see Annex E of the Base Language). Using it also for the components of a class is an additional usage, not a redefinition.

A full class definition consists of a class header followed by a class definition body. The identifier following the class definition body shall be identical to the class identifier. A forward class definition consists of a class header followed by the keyword FORWARD. Forward definitions of classes are needed to solve the declare-before-use-in-declaration problem e.g. in the case where two classes reference each other.

Concrete Syntax

```
class definition =
    ( traced class definition | untraced class definition );
untraced class definition =
    ( normal class definition | abstract class definition );
traced class definition =
    “TRACED”, ( normal class definition | abstract class definition );

normal class definition =
    normal class header, ( normal class definition body | “FORWARD” );
normal class header =
    “CLASS”, class identifier, semicolon;
normal class definition body =
    [ inherit clause ],
    [ reveal list ],
    normal class component definitions,
    “END”, class identifier;

abstract class definition =
    abstract class header, ( abstract class definition body | “FORWARD” );
abstract class header =
    “ABSTRACT”, “CLASS”, class identifier, semicolon;
abstract class definition body =
    [ inherit clause ],
    [ reveal list ],
    abstract class component definitions,
    “END”, class identifier;

class identifier = identifier;

normal class component definitions = { normal component definition };
normal component definition =
    “CONST”, { constant declaration, semicolon }           |
    “TYPE”, { type definition, semicolon }                 |
    “VAR”, { class variable declaration, semicolon }       |
    (normal method definition | overriding method definition), semicolon;

abstract class component definitions = { abstract component definition };
abstract component definition =
    “CONST”, { constant declaration, semicolon }           |
    “TYPE”, { type definition, semicolon }                 |
    “VAR”, { class variable declaration, semicolon }       |
    (normal method definition | abstract method definition | overriding method definition), semicolon;

class variable declaration = identifier list, colon, type denoter;

normal method definition = procedure heading;
overriding method definition = “OVERRIDE”, procedure heading;
abstract method definition = “ABSTRACT”, procedure heading;
```

Declaration Semantics

As in a type definition, a class definition is used to introduce an identifier for the new declared class type. As classes cannot be defined implicitly, each class type has its corresponding class identifier.

Static Semantics

If present, the inherit clause shall be valid (see 6.1.6).

If present, the reveal list shall be valid (see 6.1.5).

The identifiers declared in the class definition body shall be well-formed given the environment that applies at the point of the class definition (see also 6.1.6: inherit clause).

If a forward definition of a class is made, it shall precede a corresponding full definition of that class. The full definition shall be made in the same definition module.

If a class definition exists for a class, this class shall be declared in the corresponding implementation module. If this declaration is done in a local module, the name shall be exported unqualified to the outermost block.

NOTE 2 This rule is identical to the rule for declaring a procedure defined in a definition module.

NOTE 3 If an opaque type is defined in a class definition, it shall be declared in the corresponding class declaration.

6.1.2 Class Declaration

A class declaration defines a new structured type. Each value of the type has a collection of components, called components of the class. The components are identified by their names. The components can be accessed from the outside world via class name qualification or via qualification of an object designator (see 6.2).

A full class declaration consists of a class header followed by a class declaration body. The identifier following the class declaration body shall be identical to the class identifier. A forward class declaration consists of a class header followed by the keyword FORWARD. Forward declarations of classes are needed to solve the declare-before-use-in-declaration problem, e.g. in the case where two classes reference each other.

Concrete Syntax

```
class declaration =  
    ( traced class declaration | untraced class declaration );  
untraced class declaration =  
    ( normal class declaration | abstract class declaration );  
  
normal class declaration =  
    normal class header, ( normal class declaration body | "FORWARD" );  
normal class header =  
    "CLASS", class identifier, semicolon;  
normal class declaration body =  
    [ inherit clause ],  
    [ reveal list ],  
    normal class component declarations,  
    [ class body ],  
    "END", class identifier;  
  
abstract class declaration =  
    abstract class header, ( abstract class declaration body | "FORWARD" );  
abstract class header =  
    "ABSTRACT", "CLASS", class identifier, semicolon;  
abstract class declaration body =  
    [ inherit clause ],  
    [ reveal list ],  
    abstract class component declarations,  
    [ class body ],  
    "END", class identifier;  
  
class body = module body;
```