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)

Contents

Foreword v
Introduction vi
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 3
4.2 Source Code Representation 3
4.3 Ordering of Declarations 3
4.4 Predefined Entities 3
4.5 Library Modules 3
4.6 Errors 3
4.7 Exceptions 4
4.8 Implementation-dependencies 4
4.9 Documentation 4
4.10 Statement of Compliance 4
4.11 Minimum requirements 4
5 The Lexis 4
5.1 Additional Keywords 4
5.2 Additional Pervasive Identifiers 5
6 Language 5
6.1 Classes 6
6.1.1 Class Definition 6
6.1.2 Class Declaration 8
6.1.3 Abstract Classes 10
6.1.4 Traced Classes 10
6.1.5 Reveal Lists 12
6.1.6 Inherit Clause 12
6.1.7 Class Components 13
6.1.8 Overridden Methods 14
6.1.9 Constructor 14
6.1.10 Destructor 15

6.2 Access to Class Components....16
6.2.1 Access by Class Identifier....16
6.2.2 Access by Object Selection....16
6.2.3 Access in Subclasses....18
6.2.4 Immutable entities....18
6.3 Object Variables....18
6.3.1 Assignment....19
6.3.2 Comparison....19
6.3.3 Creation....19
6.3.4 Destruction....20
6.3.5 Empty Reference....21
6.3.6 Traced Object References....21
6.4 Membership Test....23
6.5 Guard Statement....24
6.5.1 Guarded Statement Sequence....25
6.6 Safeguarded Modules....25
6.7 Garbage Collection....25
6.8 Changes to the Base Language....26
6.8.1 Module Header....27
6.8.2 Definitions and Declarations....28
6.8.3 Blocks....29
6.8.4 Statement Part....29
6.8.5 Parameter Compatibility and Argument Binding....30
6.8.6 The Module COROUTINES....30
6.8.7 Environment....31
7 Required System Modules....31
7.1 Object Exceptions....31
7.1.1 The Interface to M2OOEXCEPTION....31
7.1.2 The Entities of M2OOEXCEPTION....31
7.1.3 Aggregation of the Exceptions of the Language Extensions....32
7.2 The Module GARBAGECOLLECTION....32
7.2.1 The interface to GARBAGECOLLECTION....32
7.2.2 The Procedures of GARBAGECOLLECTION....33
7.2.3 The Functions of GARBAGECOLLECTION....33
Annex A Collected Concrete Syntax (informative)....34
A.1 Class Definition....34
A.2 Class Declaration....35
A.3 Reveal List....36
A.4 Inherit Clause....36
A.5 Designators....36
A.6 Guard Statement....36
Annex B Changes to the Syntax of the Base Language (informative)....37
Annex C Glossary (informative)....38

Annex D Examples (informative) 41
D.1 Class Definition and Class Declaration 41
D.2 Cross Linked Classes 42
D.3 Inheritance and Overridden Methods 42
D.4 Abstract Class Definition 43
D.5 Creation and Destruction 43
D.6 Constructor and Destructor 44
D.7 Type Inquiries 44
D.8 Constructor chain and Destructor chain 45
D.9 Coroutines and Garbage Collection 45
Annex E Finalization Order (informative) 46
Annex F Participating Individuals and Organisations (informative) 48

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*

Annexes A to 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 multi-part 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.

Information technology — Programming languages —

Part 3:
Object Oriented Modula-2

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;

normal class component declarations = { normal component declaration } ;
normal component declaration =
      "CONST", { constant declaration, semicolon }              |
      "TYPE", { type declaration, semicolon }                   |
      "VAR", { class variable declaration, semicolon }          |
      normal method declarations , semicolon ;

abstract class component declarations = {abstract component declaration } ;
abstract component declaration =
      "CONST", { constant declaration, semicolon }              |
      "TYPE", { type declaration, semicolon }                   |
      "VAR", { class variable declaration, semicolon }          |
      abstract method declarations , semicolon ;

normal method declarations =
      normal method declaration | overriding method declaration;
normal method declaration = procedure declaration;
overriding method declaration = "OVERRIDE", procedure declaration;

abstract method declarations =
      normal method declaration | abstract method definition | overriding method declaration;
```

Declaration Semantics

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

Static Semantics

A class shall be declared at declaration level 0.

NOTE 1 — For the definition of "declaration level 0" see Base Language 6.2.5.

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 declaration body shall be well-formed given the environment that applies at the point of the class declaration. If a corresponding class definition exists, the starting environment of the class scope shall be the environment of the class definition instead of the empty environment (see also 6.1.6: inherit clause).

If a forward declaration of a class is made, it shall precede the corresponding full declaration of that class. The full declaration shall be made in the same block or in a nested local module from which it is exported unqualified.

A class that is exported from a separate module that has a corresponding class definition in the definition module, shall not be given a forward class declaration in the implementation module.

Dynamic Semantics

The elaboration of a class declaration shall associate, in the resulting execution environment, the identifier of the class and the unique type that is the result of the elaboration of the class declaration body.

NOTE 2 — The body of the class is executed on creation / destruction of an object of that type.

6.1.3 Abstract Classes

Abstract classes are used to define interface descriptions for a set of subclasses with a common external behaviour but different internal implementations. In an abstract class, not all methods need to be implemented, these abstract methods do have only the procedure header for the interface description and are implemented only in subclasses. Because of this incompleteness it is not possible to instantiate an object of an abstract class.

Classes may be declared as abstract by prefixing them with the keyword ABSTRACT.

NOTE 1 — Like non-abstract classes, abstract classes always need to have a declaration. If no hidden components are declared and no method has to be implemented (i.e. all methods are abstract), this declaration is empty.

Declaration Semantics

The declaration semantics of an abstract class is the same as that of a non-abstract class with the exception that the class identifier is marked abstract, so it shall not be possible to instantiate an object of that class.

NOTE 2 — No other restriction is made for the use of abstract classes. It can be used anywhere a non-abstract class can be used, especially as a type for ISMEMBER.

If the class definition is marked abstract, the class declaration also shall be marked abstract; if the class definition is not marked abstract, the class declaration shall not be marked abstract.

Static Semantics

Within an abstract class, abstract methods may be defined. They do not have an implementation. As long as there exist abstract methods, the class shall be abstract. Abstract methods are marked with the keyword ABSTRACT.

If an abstract method is defined in a class definition, there shall be no declaration for this method in the corresponding class declaration.

If there is a class definition, no abstract methods shall be specified in the corresponding class declaration, as they could not be overridden by inheritors to make the class non-abstract.

6.1.4 Traced Classes

Two kinds of class are provided: traced and untraced. This allows two kinds of object to be created: traced and untraced.

A program that uses traced objects is safe from undetected dangling object reference errors, except when the tracing mechanism is deliberately subverted. The implementation guarantees this by the automatic initialization of traced object references and the automatic collection of traced objects that are no longer referenced (see 6.3.6 and 6.7).

A program that uses untraced objects is not safe from undetected dangling object reference errors. In such a program, it is the programmer's responsibility to prevent such an error by ensuring that the program does not use uninitialized object references or references to objects that have been destroyed.

An untraced class may have a finalization body that is executed when the object is explicitly destroyed. Traced objects are automatically destroyed in an order and at a time that is implementation dependent. Therefore, a traced class shall not have a finalization body.

Concrete Syntax

```
traced class declaration =
      ( normal traced class declaration | abstract traced class declaration ) ;

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

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

traced class body = "BEGIN", block body;
```

Declaration Semantics

The declaration rules for traced classes are the same as for untraced classes, except that they are marked as traced.

Static Semantics

The static semantics of a traced class definition is identical to that for an untraced class definition.

The static semantics of a traced class declaration is identical to that for an untraced class declaration, except that it shall be an error for a traced class declaration to have a finalization body.

6.1.5 Reveal Lists

Components in a class definition / declaration are visible by default only in the class itself or in subclasses. (We term this *family* visibility, see Annex C Glossary) To expose identifiers to clients of a class the identifiers shall appear in a list of *revealed* identifiers. In a class declaration (i. e. in an implementation module) these rules also hold.

NOTE 1 — A subclass declared in the same module has access to all components of the class, as a subclass does not depend on revealing (see 6.1.6).

Concrete Syntax

reveal list = "REVEAL" revealed component list, semicolon ;

revealed component list = revealed component, { comma, revealed component } ;
revealed component = identifier | "READONLY" class variable identifier ;
class variable identifier = identifier ;

Declaration Semantics

The components listed in the reveal list are flagged to be accessible from outside the class (see 6.2).

Class variables may also be marked READONLY to restrict the access rights, preventing modification by a client (immutable entity, see 6.2.4).

Static Semantics

REVEAL is bound to the rules for the separation of definition and implementation modules. Thus, the reveal list in a class definition shall not contain components defined in the class implementation. The revealed components are accessible in all modules referring to that class as clients (by importing the module the class is defined in) and in the corresponding implementation module; i.e. components revealed in a class definition are automatically revealed in the corresponding class declaration.

The reveal list in a class declaration may contain components defined in the corresponding class definition as well as those declared in the class declaration.

The reveal list in a class declaration shall not contain components already revealed in the class definition.

The reveal list shall not contain components defined in a superclass.

NOTE 2 — Each class is owner of the access rights of its components. If for any reason a subclass wants to loosen access restrictions on a component of a superclass, it has to provide for access methods for this component.

6.1.6 Inherit Clause

Classes may inherit from at most one other class. Inheritance is specified by the new keyword "INHERIT" followed by the name of a class (which becomes the parent of the inheriting class).

Subclasses are subtypes in the following sense: if class B inherits from class A, B is a subtype of A. Object variables of type B will then be assignment compatible to object variables of type A. Object variables of type A will not be assignment compatible to object variables of type B (according to the type system of Object Oriented Modula-2, see 6.3.1). Therefore, the static type of an object variable (the one used to declare that variable) may be different from the dynamic type of the object that is referenced by this variable; the dynamic type may be a subtype.

Concrete Syntax

inherit clause = "INHERIT", class type identifier, semicolon ;
class type identifier = type identifier ;

Declaration Semantics

If class B inherits from class A, the environment for the elaboration of the entities of B shall be the same as if the already declared entities of A were actually declared in the place of the inherit clause. Inheritance does not cause nested scopes. Therefore, an identifier shall not be redefined or redeclared by an inheriting class (methods may be overridden but not redefined, see 6.1.8).

NOTE — As a result, class B contains all entities of A plus all the additional entities of B.

Static Semantics

If a class definition exists, inheritance may only be specified in the class definition. Conversely, inheritance may only be used in a class declaration if no corresponding class definition exists.

It shall be an error for a traced class to inherit from an untraced class. It shall be an error for an untraced class to inherit from a traced class.

6.1.7 Class Components

The elaboration of class component definitions/class component declarations acts similarly to the elaboration of definitions/declarations described in the base language. There are some differences with class variables and methods that are described in this subclause.

Declaration Semantics

Class variables:
A class variable is collected to the type structure of the class at declaration time (like a field of a record to the type structure of that record); at run-time this structure is used to allocate storage for each object.
Methods:
Each method definition / declaration establishes an implicit first formal parameter with the parameter identifier SELF; the type of SELF shall be a reference to the class the method is declared in.
SELF shall denote an immutable entity (see 6.2.4)

NOTE — The fact that SELF is the first parameter with respect to the formal elaboration has no implication on the implementation, e. g. in what position this parameter is actually passed.

Dynamic Semantics

Class variables:
No storage is allocated for a class variable (see above).
Methods:
A call to a method automatically binds the object reference used to invoke the call (see 6.2.2) to the implicit first parameter SELF.

6.1.8 Overridden Methods

Methods of an ancestor class A can be overridden by a direct or indirect descendant class B. Overridden methods preserve the method interface but replace the implementation. Method calls (via dynamic object access) will always refer to the method implementation according to the dynamic type of the object.

Declaration Semantics

By overriding an existing method, no new component is established. The meaning of the overridden method is replaced by the meaning of the overriding method.

For the overriding method, the type of the implicit parameter SELF, is the type of the class in which this method is overridden.

Static Semantics

To be able to override an existing inherited method, the method definition/declaration of the subclass containing the override shall be marked by the new keyword OVERRIDE.

If a class definition exists, OVERRIDE shall be used in both the class definition and the class declaration if the method is to be overridden. Failing to do either of these will result in a compilation error.

A method that is marked as overridden shall exist in an ancestor class; otherwise a compilation error shall occur.

The overriding method shall have the same number, positions, and types of parameters as the overridden method; otherwise a compilation error shall occur. If the overridden method is a function procedure method, the overriding method shall also be a function procedure method and additionally to the restriction on the parameters the result types shall be identical.

6.1.9 Constructor

A constructor of an object is code that is executed during its creation. A class declaration may contain a class body (which has the same syntax as a module body) in which the initialization part of the body plays the role of a constructor.

Declaration Semantics

As with the elaboration of a method declaration, the elaboration of the initialization part of a class body shall establish the implicit parameter SELF, usable inside the initialization part as a reference to the object under construction. The initialization body shall be added to the (possibly empty) constructor chain.

NOTE 1 — The construction of the constructor chain is similar to the construction of the initialization chain of modules; the import relationship is replaced by the inherit relationship.

Static Semantics

The environment of the initialization part shall be that established at the beginning of the class body, overwritten by the (implicit) declaration of the formal parameter SELF.

Dynamic Semantics

The constructor is automatically invoked on creation of an object of the given class (see 6.3.3).

For inheritance chains, the execution order of the constructors is from root class to leaf class, i.e. the constructor of a superclass is executed before the constructor of a subclass. The execution of the constructor of a superclass is not a call from within the actual constructor but acts like the initialization of dynamic modules within a procedure. In particular, an exception handler within a constructor of a class does not protect the execution of the constructor of a superclass.

Constructors are executed in their static context, i. e. all method calls during construction time are done without dynamic dispatch. On a call to an abstract method, an exception shall occur whose detection is mandatory (see 7.1.3).

NOTE 2 — No dynamic overhead is caused by this check, as it can be done within a default body for abstract procedures.

6.1.10 Destructor

A destructor of an object is code that is executed during the destruction of the object. A class declaration may contain a class body in which the finalization part of the body plays the role of a destructor.

Declaration Semantics

As with the elaboration of a method declaration, the elaboration of the finalization part of a class body shall establish the implicit parameter SELF, usable inside the finalization part as a reference to the object under destruction. The finalization body shall be added to the (possibly empty) destructor chain.

Static Semantics

The environment of the finalization part shall be that established at the beginning of the class body, overwritten by the (implicit) declaration of the formal parameter SELF.

Dynamic Semantics

The destructor is automatically invoked on destruction of an object of the given class (see 6.3.4).

For inheritance chains, the execution order of the destructors is from leaf class to root class, i.e. the destructor of a superclass is executed after the destructor of a subclass. The execution of the destructor of a superclass acts like the finalization of dynamic modules within a procedure. In particular, an exception handler within a destructor does not protect the execution of the destructor of a superclass.

NOTE — Whereas the constructor is executed in its static context, the destructor is executed in the dynamic context.

6.2 Access to Class Components

Inside a class, all components are visible according to the scope rules. This is also true for components of superclasses (with exception of overridden methods). Therefore, no special syntax is required; the meaning of the access to attributes and methods is described later in this subclause.

For users of a class (the clients), access to class entities from outside the class scope is provided by qualifying either a class identifier or an object variable. Only components marked as "REVEAL" are accessible from outside.

6.2.1 Access by Class Identifier

Access by class identifier acts like the qualification by a module name. As in this case there is no object for an attribute to belong to or for a method to act on, only type identifiers and constant identifiers (including identifiers specifying the values of an enumeration type) can be accessed in this way.

Concrete Syntax

qualified identifier =
    { qualifying identifier, period }, [class identifier, period], identifier ;

Static Semantics

If a class identifier is provided, the identifier is looked for in the scope of the class specified by the given class identifier. No class variable or method can be accessed from outside the class. No unrevealed component shall be accessed from outside the class. If no class identifier is provided, the definition of the base language holds.

NOTE — As the components defined in a superclass become part of the scope of a subclass, the identifier may be defined either in the given class or in a superclass.

6.2.2 Access by Object Selection

Access by object selection acts like selection of a record field. With object selection, all components of a class are accessible from inside the class, all revealed components are accessible from outside the class. For class inheritance trees, it is possible to specify the class where the selected identifier shall be looked for.

NOTE 1 — Though in principle allowed by the syntax of selection, access to a type identifier declared in a class by object selection is impossible because of the general syntax governing the rules of the use of types.

Value designator and variable designator are described together, as constant object references shall not occur (see static semantics). The rules given in the base language describe the meaning of the selected components, the access of the values of the selected components, and the assignment to the selected components.

Concrete Syntax

```
variable designator = entire designator | indexed designator | selected designator |
      dereferenced designator | object selected designator ;
object selected designator = object variable designator, period, [ class identifier, period ],
      class variable identifier ;
object variable designator = variable designator ;

value designator =
      entire value | indexed value | selected value | dereferenced value | object selected value ;
object selected value =
      object value designator, period, [ class identifier, period ], entity identifier ;
object value designator = value designator ;
entity identifier = identifier ;
```

Static Semantics

The object variable designator/object value designator shall denote an object variable/value. The declaration of constant object values shall be an error, i. e. SYSTEM. CAST shall not be enhanced to allow a class type as first parameter and a constant as second parameter.

The type of an object variable designator / object value designator shall be the type of the component identifier.

If a class identifier is specified, it shall denote the class of the object variable or a superclass of this class. In this case, the selected identifier shall be looked for in the specified class.

NOTE 2 — Specifying a class is used to call to an overridden method. In all other cases, it has no special functionality; it only verifies that the entity is available in the specified class scope.

NOTE 3 — The access to a constant by object selection is not a constant expression.

Dynamic Semantics

The object variable designator shall be evaluated to give a variable P which is a reference to an object of the type of the object variable .

An exception shall be raised, if the object reference is the empty reference (see 7.1.3).

An exception shall occur (but need not be raised), if the object reference is undefined.

An exception shall occur (but need not be raised), if the object reference is a reference to the wrong class, i.e. not to the class given by the object variable or a descendant class of this class.

The exception that is to be raised in case of an empty reference is defined in subclause 7.1.3; the exceptions that may be raised in the other two cases are those defined in the base language.

Access to a constant by object selection gives the value of this constant.

Access to a class variable by object selection gives the value stored in this class variable.

Access to a method by object selection always results in a procedure call. The meaning of procedure calls and the meaning of argument binding as defined in the base language apply with the addition, that the object reference used for the access to the method becomes the value for the first implicit parameter (SELF).

6.2.3 Access in Subclasses

As all components of a class are visible in subclasses (except hidden components, if the subclass is declared in another compilation unit), no class specification/object variable is needed to access the components of a class. For constants and types, the normal rules apply.

For the access to class variables and methods (which can only occur in the statement part of a method or in the class body), the implicit parameter SELF defines the object the class variable belongs to or the method it acts on.

The optional specification of the class identifier of a superclass is also possible for the access to class components within a subclass (see concrete syntax of qualified identifier). In this case, the component is looked for in the given scope (as described in the previous subclauses). This kind of access is used to access overridden methods that are no longer accessible directly in the subclass.

6.2.4 Immutable entities

Entities that shall not be changed are called immutable entities. Immutable entities occurring in this standard are:
— the implicit SELF parameter of methods, constructors and destructors,
— the class variables marked readonly when used by a client,
— the object denoters declared in the object denoter part of a guarded statement sequence.

Static Semantics
Immutable entities shall not occur on the left hand side of an assignment statement or as actual parameter for a formal VAR-parameter.

NOTE 1 — This also prevents "threatening" by SYSTEM.ADR.

Dynamic Semantics
An exception shall occur (but need not be raised), if an immutable entity is changed (see 7.1.3).

NOTE 2 — A class variable marked readonly is immutable only when accessed by a client; it may well be changed from within a method of the defining class and its descendants.

6.3 Object Variables

Object variables always store references to objects. Any operation on object variables is an operation on the object reference, not on the object itself. Therefore, objects shall be explicitly created and either explicitly destroyed (untraced classes) or implicitly destroyed by garbage collection (traced classes).

6.3.1 Assignment

The assignment compatibility rules and the meaning of the assignment statement are extended for objects. Objects are not only assignment compatible to object variables of the same class but also to object variables of any ancestor class. EMPTY is assignment compatible to any object variable.

Static Semantics

The type Tv of an object variable denoted by a variable designator shall be assignment-compatible with the type T_e of the value of an expression, if any of the following statements is true:
— Tv and T_e are identical types.
— T_e is a subclass of Tv.
— T_e is the empty type.

Dynamic Semantics

The execution of an object variable assignment statement shall cause the object reference value resulting from the elaboration of the expression to become the new value of the object variable denoted by the variable designator.

All rules defined for the elaboration of an assignment statement as defined in the Base Standard also apply for object variable assignments.

6.3.2 Comparison

The relational operators are overloaded for object reference comparison. Two comparison operators are provided, test for equality ("=") and test for inequality ("<>" or "#"). However, both operators compare object references, not objects themselves.

Static Semantics

Both operands of an object reference comparison shall be of object types. One of the following shall hold:
— The type of the left operand is assignment-compatible to the type of the right operand.
— The type of the right operand is assignment-compatible to the type of the left operand.
— Both operands denote the empty reference.
The result of an object reference comparison shall be of the Boolean type.

Dynamic Semantics

The evaluation of the left and right expression of an object reference comparison shall result in two values which are references to objects.
The value of an object reference equality operation shall be the value *true* if and only if the two references are identical.
The value of an object reference inequality operation shall be the value *true* if and only if the two references are different.

6.3.3 Creation

Objects are allocated storage and thereby instantiated by a call to the new predefined procedure CREATE, a template for which is given below:

```
CREATE (variable [, type])
```

Static Semantics

"variable" shall denote a variable designator which is of class type.

The second parameter, if present, shall be a type parameter that denotes a class type which is assignment compatible to the type of the first parameter.

The dynamic type of the new object is either the statically declared class type of "variable" or, if present, the class type specified by the second parameter. That is, a variable statically declared to be of one class type could be dynamically created with a subclass type.

If the type of the first parameter denotes an untraced class, the identifier "ALLOCATE" shall be visible as required by the pervasive procedure "NEW".

Dynamic Semantics

If the first parameter to a call to CREATE is a variable of the traced class type, then use *allocate-traced-object-storage* to allocate storage for the object. For an untraced object, storage is allocated by a call to ALLOCATE.

If storage cannot be allocated for the object, the value EMPTY shall be assigned to "variable", otherwise a reference to the new object is stored in "variable".

NOTE — No exception occurs in the case of an unsuccessful storage allocation.

If storage allocation was successful,:

— If the first parameter to a call to CREATE is a variable of a traced class type, then the garbage collector shall be informed of a *new-traced-variable* in respect of the new value and then informed of a *defunct-traced-variable* in respect of the old value (see 6.7).

— The constructor chain of the new object shall be executed.

6.3.4 Destruction

Invoking the destruction of an object is different for traced and untraced objects.

Traced objects are implicitly destroyed by garbage collection, if doing so will leave no dangling references to them (see 6.7).

Untraced objects are explicitly destroyed by a call to the new predefined procedure DESTROY, a template for which is given below:

```
DESTROY (variable)
```

Static Semantics

"variable" shall denote a variable designator which is of object type of an untraced class.

The identifier DEALLOCATE shall be visible as required by the pervasive procedure DISPOSE.

It shall be an error for the parameter to a call of DESTROY to be a variable of the traced class type.

Dynamic Semantics

If "variable" contains the empty reference, an exception shall occur and shall be raised as for any other access to an object via the empty reference (see 6.3.5).

The destructor chain of the object shall be invoked, i.e. the destructors are elaborated in the reverse order of the constructors (see 6.1.10).

The storage occupied by the object shall be returned by a call to DEALLOCATE.

NOTE — For the call to DEALLOCATE, the dynamic type of the object (and the space it occupies) is detected by the runtime system.

6.3.5 Empty Reference

A new constant, denoted by the new pervasive identifier EMPTY, is introduced and denotes the non-existing object. The type of EMPTY is the *empty type*. The value of EMPTY shall be an object reference distinct of all references to existing objects, it is referenced as *empty reference* within this document.

EMPTY is compatible with all object references (see 6.3.1 and 6.3.2).
Any attempt to access an object via an object variable whose value is the empty reference shall be an exception whose detection is mandatory (see 7.1.3).

6.3.6 Traced Object References

Traced object references in global, local and class variables are initialized to EMPTY by the implementation. This ensures that dangling object reference errors caused by the use of object references that do not point to valid objects will raise an exception.

The programmer is not permitted to destroy a traced object. The implementation is permitted to destroy a traced object when doing so will leave no dangling references to it. The implementation does this by tracking references to traced objects stored in global and local variables and parameters. This ensures that dangling object reference errors caused by the destruction of an object that has more that one reference to it can not occur.

NOTE 1 — The programmer indicates that a traced object can be destroyed by assigning to all variables or parameters that reference the object the value EMPTY or a reference to another traced object.

NOTE 2 — An implementation is not required to track references to traced object stored in dynamic variables or obtained by the use of procedures from the system module SYSTEM.

NOTE 3 — Whether and when an implementation destroys a traced object is implementation dependent.

6.3.6.1 Initialization of Object Referencing Variables

Dynamic Semantics

Global elementary variables of a traced class type and components of global structured variables of a traced class type shall be automatically initialized with the value EMPTY as each static module and the program module is elaborated (see 6.8.2).

NOTE 1 — This includes the initialization of variables declared in local modules.

NOTE 2 — Initialization of these variables is performed before any finalization body is queued.

Local elementary variables of a traced class type and components of local structured variables of a traced class type shall be initialized with the value EMPTY on activation of a procedure (see 6.8.2).

NOTE 1 — This includes the initialization of variables declared in local (dynamic) modules.

NOTE 2 — Initialization of variables is performed before any local modules are initialized.

Elementary class variables of a traced class type and components of structured class variables of a traced class type shall be initialized with the value EMPTY on creation of an object.

Class variables shall be initialized before the constructor chain of a new object is executed.

NOTE 3 — It is not permitted to declare a variant field of a record to be a traced class type (see 6.8.2).

NOTE 4 — It is not permitted to declare a variable that points to a traced object reference (see 6.8.2).

6.3.6.2 Creation of an Object Reference

Dynamic Semantics

When an object of a traced class type is created the garbage collector shall be informed of the object and of the variable to which the initial reference was assigned (see 6.3.3).

NOTE 1 — It is not permitted to declare a variable that points to a traced object reference (see 6.8.1).

When a traced object reference is bound to a value formal parameter the garbage collector shall be informed that the object referred to is referenced by the formal parameter (see 6.8.5 and 6.8.4).

When a structured value that contains at least one traced object reference is bound to a value formal parameter the garbage collector shall be informed that the objects referred are referenced by the formal parameter components (see 6.8.5 and 6.8.4).

When a variable actual parameter to a procedure call is the designator for a class variable of an object of a traced class type the garbage collector shall be informed that the object is implicitly referenced by the parameter.

NOTE 2 — It is not necessary to trace the subsequent passing of such a formal parameter via a variable parameter because it is protected from collection by the traced reference created by the initial call.

6.3.6.3 Changing an Object Reference

Dynamic Semantics

When a new value that is not EMPTY is assigned to an elementary variable of a traced class type the garbage collector shall be informed that the variable now points to the object referenced by the new value. If the previous value was not EMPTY, then the garbage collector shall be informed that the variable no longer references the object previously referred to by the variable (see 6.8.4).

When a new value that is not EMPTY is assigned to a component of a structured variable that is of a traced class type the garbage collector shall be informed that the variable's component now points to the object referenced by the new value. If the previous value was not EMPTY, then the garbage collector shall be informed that the variable's component no longer references the object previously referred to by the component (see 6.8.4).

NOTE 1 — It is not permitted to declare a variable that points to a traced object reference (see 6.8.1).

NOTE 2 — It is not permitted to declare a variant field of a record to be of a traced class type (see 6.8.1).

6.3.6.4 Deletion of an Object Reference

Dynamic Semantics

On completion of the execution of a procedure block or if an exception is raised during the execution of the body of a procedure block that has no exceptional part, or if an exception is raised during the execution of the exceptional part of a procedure block, the following shall be performed:

— The garbage collector shall be informed that local elementary variables of a traced class type and components of local structured variables of a traced class type that do not contain the value EMPTY no longer contain an active reference (see 6.8.3).

— The garbage collector shall be informed that value formal parameters of a traced class type that do not contain the value EMPTY no longer contain an active reference (see 6.8.2).

— The garbage collector shall be informed that components of structured value formal parameters of a traced class type that do not contain the value EMPTY no longer contain an active reference (see 6.8.2).

— The garbage collector shall be informed that the variable parameters to the procedure call that were designators for a class variable of a traced class type no longer contain an active reference.

On destruction of an untraced object the garbage collector shall be informed of a *defunct-traced-variable* for each class variable of the object that is an elementary variable of a traced class type or component of a structured variable of a traced class type that does not contain the value EMPTY.

NOTE — It is not necessary for the garbage collector to be informed of a *defunct-traced-variable* for the class variables of a traced object because this is implicit in the object becoming collectable (see 6.7).

At the end of the lifetime of a coroutine, for each active traced object reference stored in the coroutine's workspace the garbage collector shall be informed that the reference is now defunct (see 6.8.6).

6.4 Membership Test

The new pervasive function procedure ISMEMBER is introduced to determine the dynamic type of an object.

Static Semantics

ISMEMBER takes two parameters, which may be either object value designators or type parameters of class type. The result type of ISMEMBER is the Boolean type.

Dynamic Semantics

If not a type parameter, the parameters are evaluated and the dynamic type of each is determined. ISMEMBER returns true, if and only if the (dynamic) type of the first parameter is a descendant of or equal to the (dynamic) type of the second parameter.

NOTE — If the first parameter is an object and the second is a class type name, ISMEMBER returns true if the object is assignment compatible to the specified class.

6.5 Guard Statement

A new guarded region statement is introduced (keyword GUARD) to provide an efficient combination of type assertion (in which the dynamic type is asserted to be compatible with a static type) and type selection.

Concrete Syntax

```
statement = ... | guard statement;

guard statement =
      "GUARD", guard selector, "AS", guarded list,
      ["ELSE" statement sequence],
      "END";

guard selector = expression ;

guarded list = guarded statement sequence {vertical bar, guarded statement sequence} ;
guarded statement sequence =
      [[object denoter], colon, guarded class type, "DO", statement sequence] ;
guarded class type = class type identifier ;
object denoter = identifier ;
```

Static Semantics

The guard selector shall be a well formed expression of a class type.

Dynamic Semantics

When executing the guard statement, the guard selector is evaluated. The dynamic type of the selector object is checked against the class types and the first (in lexical order) matching guarded statement sequence is selected. The dynamic type of the guard selector object matches a guarded class type if it is assignment compatible to that type but not the empty reference, i. e. it is of the guarded class type or one if its subclasses.

If the selector object does not match any of the guarded statement sequences, the ELSE part of the guard statement shall be executed (if present); otherwise an exception shall be raised (see 7.1.3).

NOTE — The empty reference always selects the ELSE part (or raises the exception).

6.5.1 Guarded Statement Sequence

Within a selected guarded statement sequence, the selector object is accessible via the object denoter (if present).

Static Semantics

The guarded statement sequence opens a new scope (like the with statement), the object denoter, if present, is declared in this scope. The object denoter is an immutable entity inside the statement sequence and has the static type as specified by the guarded class type. The statement sequence shall be well-formed within this scope.

Dynamic Semantics

If the guarded statement sequence is selected, the object reference that results from the evaluation of the guard selector is assigned to the object denoter (if present). Then the statement sequence is executed.

6.6 Safeguarded Modules

Unless a compilation module is tagged as unsafeguarded it shall contain neither untraced objects nor statements that threaten the ability of the garbage-collector to track references to traced objects. A compilation module is tagged as unsafeguarded by including the keyword UNSAFEGUARDED in its module header (see 6.8.1).

If a program contains no unsafeguarded modules then it is guaranteed to be safe from undetected dangling object reference errors. A program that contains unsafeguarded modules is no longer safe from such errors. In such a program, it is the programmer's responsibility to ensure that the unsafeguarded modules do not cause such errors.

NOTE 1 — Even in a program that contains no unsafeguarded modules, dangling object reference errors can still be caused by injudicious use of features from the SYSTEM module.

NOTE 2 — Modules that do not use object oriented extensions are safeguarded by definition.

6.7 Garbage Collection

The garbage collector identifies and has the option to destroy traced objects that are no longer referenced. Along with the automatic initialization of traced object references this ensures object safety by guaranteeing not to destroy an object that is referenced by a global, local or class variable or by a procedure parameter. It does this by tracking the references to a traced object and only destroying it when to do so will leave no dangling references to it.

NOTE 1 — An implementation is not required to track references to traced objects obtained by the use of procedures from the system module SYSTEM. If such untracked references are the only references to a traced object the implementation is allowed to collect the object.

NOTE 2 — It is not the intention of this standard to specify the way traced object references shall be tracked. An implementation is free to use any scheme that has the same behaviour as the one specified.

The garbage collector allocates and deallocates storage for traced objects.

NOTE 3 — This standard does not provide a means (for a user of an implementation) to replace the storage management scheme employed for traced objects.

Dynamic Semantics

This specification models the behaviour of the garbage collector in terms of operations on a map from objects to sets of references. The operations refer to flag *garbage-collection-enabled* which is maintained by procedures of the system module GARBAGECOLLECTION (see 7.2). Before module initialization, *garbage-collection-enabled* is set to be true, and the map is set to be empty.

When a variable is set to refer to a newly created object, the object shall be added to the map in association with a singleton set containing the variable. Such an operation is a *new-traced-variable*.

When a reference to a non-empty object is made, the referring variable shall be included in that set of the map which is associated with the object. Such an operation is an *additional-traced-variable*.

When a reference to a non-empty object ends, the referring variable shall be excluded from that set of the map which is associated with the object. Such an operation is a *defunct-traced-variable*.

At times that are implementation dependent, but subject to the truth of *garbage-collection-enabled*, the garbage collector shall attempt to reclaim object storage.

NOTE 4 — This specification does not oblige an implementation to attempt reclamation at any particular time, or even at all.

When the garbage collector is reclaiming object storage it shall only reclaim the memory occupied by an object if either of the following is true:

— the set of references associated with the object is empty

— all variables in the object's set of references are components of objects whose associated sets contain only variables that are components of the object in question, or variables that are directly or indirectly components of such objects.

NOTE 5 — This specification allows an object to be collected when the only remaining active references to that object are contained within objects that may also be collected.

NOTE 6 — This specification does not oblige an implementation to reclaim all, or even any, unreferenced objects. It therefore allows a garbage collector to be "leaky".

When the garbage collector reclaims the memory occupied by an object it removes the object from the map and uses *deallocate-traced-object-storage*.

When storage is required for a traced object it shall be allocated in an implementation dependent manner. Such an operation is an *allocate-traced-object-storage*. If the requested object storage cannot be allocated the value EMPTY is returned.

When storage is no longer required for a traced object it shall be deallocated in an implementation dependent manner. Such an operation is a *deallocate-traced-object-storage*.

6.8 Changes to the Base Language

This subclause describes, where changes in the definition of the base language have to be made in order to incorporate the extensions described in this standard. A collection of the changes to the concrete syntax is found in Annex B.

6.8.1 Module Header

Program Modules

Concrete Syntax

The concrete syntax is changed to allow the module to be tagged as unsafeguarded by permitting the optional keyword UNSAFEGUARDED to appear before the keyword MODULE (Base Language 6.1.2).

Static Semantics

The following test is added to the well-formed rules for the module (Base Language 6.1.2):

Unless the module is tagged as unsafeguarded in its header, it shall contain none of the following:

— An import that directly or indirectly causes the import of an untraced class type.

NOTE 1 — This includes the import of
— an untraced class type
— a structured type that contains a component of an untraced class type
— a procedure type that returns a value of an untraced class type
— a procedure type that returns a structure that contains a component of an untraced class type or the import of any variable or procedure that is of such a type.

— A declaration of an untraced class type.

— A call of SYSTEM.CAST with a variable of a traced class type or a structured type that contains a component of a traced class type as the second actual parameter.

— A call that passes a variable of a traced class type or a structured type that contains a component of a traced class type as a value actual parameter to an open or fixed ARRAY OF LOC.

— A call of SYSTEM.ADR with a class variable (attribute) as the actual parameter.

— A call that passes a class variable (attribute) as a variable actual parameter to an open or fixed ARRAY OF LOC.

Definition Modules

Concrete Syntax

The concrete syntax is changed to allow the module to be tagged as unsafeguarded by permitting the optional keyword UNSAFEGUARDED to appear before the keyword DEFINITION (Base Language 6.1.3).

Static Semantics

The following test is added to the well-formed rules for the module (Base Language 6.1.2):

Unless the module is tagged as unsafeguarded in its header, it shall contain none of the following:

— An import that directly or indirectly causes the import of an untraced class type.

NOTE 2 — This includes the import of
— an untraced class type
— a structured type that contains a component of an untraced class type
— a procedure type that returns a value of an untraced class type
— a procedure type that returns a structure that contains a component of an untraced class type or the import of any variable or procedure that is of such a type.

— A definition of an untraced class type.

Sourced Implementation Modules

Concrete Syntax

The concrete syntax is changed to allow the module to be tagged as unsafeguarded by permitting the optional keyword UNSAFEGUARDED to appear before the keyword IMPLEMENTATION (Base Language 6.1.4.1).

Static Semantics

The following test is added to the well-formed rules for the module (Base Language 6.1.4.1):

Unless the module is tagged as unsafeguarded in its header, it shall contain none of the following:

— An import that directly or indirectly causes the import of an untraced class type.

NOTE 3 — This includes the import of
— an untraced class type
— a structured type that contains a component of an untraced class type
— a procedure type that returns a value of an untraced class type
— a procedure type that returns a structure that contains a component of an untraced class type or the import of any variable or procedure that is of such a type.

— A declaration of an untraced class type.
— A call of SYSTEM.CAST with a variable of a traced class type or a structured type that contains a component of a traced class type as the second actual parameter.
— A call that passes a variable of a traced class type or a structured type that contains a component of a traced class type as a value actual parameter to an open or fixed ARRAY OF LOC.
— A call of SYSTEM.ADR with a class variable (attribute) as the actual parameter.
— A call that passes a class variable (attribute) as a variable actual parameter to an open or fixed ARRAY OF LOC.

NOTE 4 — If a definition module is unsafeguarded its implicit import into its implementation module shall require that module to be tagged as unsafeguarded .

6.8.2 Definitions and Declarations

Definitions

The "class definition" is added to the description of "definition".

Declarations

The "class declaration" is added to the description of "declaration".

Qualified Identifier

The possibility of qualifying a class identifier is added to the description of "qualified identifier".

Variant Records

It shall be an error to declare a type that is or contains a record with a variant of a traced class type (Base Language 6.2.9, Type Declaration).

It shall be an error to declare a variable of a type that is or contains a record with a variant of a traced class type (Base Language 6.2.10, Variable Declaration).

Pointer Types

It shall be an error to declare a pointer type with a bound type that is a traced class type or a structured type with a component that is of a traced class type (Base Language 6.2.9, Type Declaration).

It shall be an error to declare a variable of a pointer type with a bound type that is a traced class type or a structured type with a component that is of a traced class type (Base Language 6.2.10, Variable Declaration).

Variable Allocation

Add in *allocate-a-variable* before the return perform (Base Language 6.2.10, Variable Declaration):

If the variable is of a traced class type, then assign the value EMPTY to it. If the variable is structured, then assign the value EMPTY to each of its components that are of a traced class type.

Parameter Deallocation

Add in *deallocate-parameters* before *deallocate-storage* perform (Base Language 6.2.11.1, Proper Procedure Declarations, Auxiliaries):

If the parameter is of a traced class type, then the garbage collector shall be informed of a *defunct-traced-variable* (see 6.7). If the parameter is structured, then for each component that is of a traced class type the garbage collector shall be informed of a *defunct-traced-variable* in respect of each object referred to (see 6.7).

6.8.3 Blocks

Add in *deallocate-declarations* before *deallocate-storage* perform (Base Language 6.5.1 Proper Procedure Blocks, Auxiliaries).

If the variable is of a traced class type, then the garbage collector shall be informed of a *defunct-traced-variable* (see 6.7). If the variable is structured, then for each component that is of a traced class type the garbage collector shall be informed of a defunct-traced-variable in respect of each object referred to (see 6.7).

6.8.4 Statement Part

Statement

The "guard statement" is added to the alternatives defined for "statement".

Assignment

Additions have to be made to the rules for assignment compatibility and to the meaning of the assignment statement to cover the assignment of object references.

Add in *elementary-variable-assignment* after both instances of *change-value* perform (Base Language 6.6.3 Assignment Statement, Dynamic Semantics):

If the assignment is to a variable of a traced class type, then the garbage collector shall first be informed of an *additional-traced-variable* in respect of the new value and then informed of a *defunct-traced-variable* in respect of the old value (see 6.7).

Relational Operations

The rules for the comparison of object references have to be added to the rules for relational operations.

6.8.5 Parameter Compatibility and Argument Binding

Add in *bind-value* before *assign* perform (Base Language 6.9.4.1 Argument Binding to Value Parameters, Dynamic Semantics):

If the parameter is of a traced class type, then assign the value EMPTY to it. If the parameter is structured, then assign the value EMPTY to each of its components that are of a traced class type.

6.8.6 The Module COROUTINES

Change in the description of the pseudo module COROUTINES (Base Language 7.2.1 The Interface to COROUTINES)

Add the following to the pseudo-definition module:

```
PROCEDURE COROUTINEDONE (cr: COROUTINE);
  (* Asserts that the coroutine identified by cr has reached
     the end of its lifetime.
  *)
```

Declaration Semantics

Add COROUTINEDONE to map as a *COROUTINE-PROPER-PROCEDURE*.

New Base Language 7.2.3.7 The Procedure COROUTINEDONE

The procedure COROUTINEDONE is used by the programmer to assert that a coroutine has reached the end of its lifetime, and that no further transfer to the coroutine will occur. This informs the garbage collector that any active traced object references contained in the workspace of the coroutine are now defunct.

NOTE — A call of COROUTINEDONE does not prevent a later transfer to the specified coroutine. Such a transfer can, however, result in erroneous operation, and is analogous to transferring to a coroutine whose workspace is no longer available.

Static Semantics

A call of COROUTINEDONE shall have one actual parameter that shall be an expression that is of the coroutine type.

Dynamic Semantics

The call COROUTINEDONE(cr) shall inform the garbage collector of a *defunct-traced-variable* for each active traced object reference in the workspace of the coroutine identified by cr.

6.8.7 Environment

Structures

The class type as described in subclause 6.1 is denoted as a type structure called *Class-structure*. This type structure *Class-structure* is added to the list of alternatives for the VDM type *Structure* (Base Language 6.11.1.3, Structures).

NOTE — Class types are not a substructure of any other structured type as defined in the Base Language. This assures that none of the features of the Base Language defined for special groups of types (e. g. type conversion) applies to class types. This also states that pointers and class references are not compatible to each other, as all compatibility rules defined in the Base Language (expression compatibility, assignment compatibility, and parameter compatibility) refer either to equal types or to types especially named in these compatibility rules.

7 Required System Modules

Two new system modules are provided for the OO extensions, one for the handling of the additional exceptions and one for the garbage collector management.

7.1 Object Exceptions

The system module M2OOEXCEPTION provides facilities for identifying language exceptions that have been raised by using features described in this standard.

7.1.1 The Interface to M2OOEXCEPTION

The interface to M2OOEXCEPTION behaves as if the following were its definition module.

```
DEFINITION MODULE M2OOEXCEPTION;

(* Provides facilities for identifying exceptions of the extended language
*)

TYPE
  M2OOExceptions =
    (emptyException, abstractException, immutableException, guardException
    );

PROCEDURE M2OOException (): M2OOExceptions;
  (* If the current coroutine is in the exceptional execution state because
     of the raising of an exception of the language extensions, returns the
     corresponding enumeration value, and otherwise raises an exception.
  *)

PROCEDURE IsM2OOException (): BOOLEAN;
  (* If the current coroutine is in the exceptional execution state because
     of the raising of an exception of the language extensions, returns
     TRUE, and otherwise returns FALSE.
  *)

END M2OOEXCEPTION.
```

7.1.2 The Entities of M2OOEXCEPTION

All entities defined in the module "M2OOEXCEPTION" behave as the corresponding entities of the module "M2EXCEPTION" of the base language, except that they act on the enumeration value "M2OOExceptions".

7.1.3 Aggregation of the Exceptions of the Language Extensions

The extended language defines four new exceptions, for three of them the detection is mandatory ("emptyException", "abstractException", "guardException"); for the forth the detection is non-mandatory ("immutableException").

— emptyException
to be raised whenever an attempt is made to access an object via an object variable that contains the empty reference.
— abstractException
to be raised whenever an attempt is made to call to an abstract method.
— immutableException
may be raised on an attempt to change an immutable entity.
— guardException
to be raised whenever the selector object in a guard statement does not match any of the guarded statement sequences and no ELSE part is present.

NOTE — A call to an abstract method can occur only during construction time of an object.

The messages associated with these exceptions are implementation defined.

7.2 The Module GARBAGECOLLECTION

The system module GARBAGECOLLECTION provides facilities for controlling the garbage collection process.

7.2.1 The interface to GARBAGECOLLECTION

The interface to GARBAGECOLLECTION behaves as if the following were its definition module.

```
DEFINITION MODULE GARBAGECOLLECTION;

(* Provides facilities for controlling the garbage collector.
*)

PROCEDURE IsCollectionEnabled (): BOOLEAN;
  (* If garbage collection is enabled then returns TRUE and otherwise
     returns FALSE.
  *)

PROCEDURE SetCollectionEnable (on: BOOLEAN);
  (* If on is TRUE then enable garbage collection; otherwise if on is
     FALSE and garbage collection can be disabled then disable garbage
     collection.
  *)

PROCEDURE ForceCollection;
  (* If garbage collection can be forced then force it else do nothing.
  *)

END GARBAGECOLLECTION.
```

7.2.2 The Procedures of GARBAGECOLLECTION

7.2.2.1 The Procedure SetCollectionEnable

Static Semantics

A call of SetCollectionEnable shall have one actual parameter which shall be an expression of the Boolean type.

Dynamic Semantics

Whether or not garbage collection can be disabled shall be implementation defined. If the implementation does not allow garbage collection to be disabled then the call SetCollectionEnable (on) shall have no action; otherwise the call SetCollectionEnable(on) shall assign the value true to the flag *garbage-collection-enabled* if the value on is TRUE, otherwise it shall assign the value FALSE to the flag *garbage-collection-enabled*.

7.2.2.2 The Procedure ForceCollection

Static Semantics

A call of ForceCollection shall have no actual parameters.

Dynamic Semantics

Whether or not garbage collection can be forced shall be implementation defined. If the implementation does not allow garbage collection to be forced then the call ForceCollection() shall have no action; otherwise the call ForceCollection() shall attempt to reclaim object storage (see 6.7).

7.2.3 The Functions of GARBAGECOLLECTION

7.2.3.1 The Function IsCollectionEnabled

Static Semantics

A call of IsCollectionEnabled shall have no actual parameters. The type of a call of IsCollectionEnabled shall be the Boolean type.

Dynamic Semantics

If the value of the flag *garbage-collection-enabled* is true, the value of the call IsCollectionEnabled() shall be TRUE, otherwise the value shall be FALSE.

Annex A
(informative)

Collected Concrete Syntax

A.1 Class Definition

```
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;
```

A.2 Class Declaration

```
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;

normal class component declarations = { normal component declaration } ;
normal component declaration =
      "CONST", { constant declaration, semicolon }              |
      "TYPE", { type declaration, semicolon }                   |
      "VAR", { class variable declaration, semicolon }          |
      normal method declarations , semicolon ;

abstract class component declarations = {abstract component declaration } ;
abstract component declaration =
      "CONST", { constant declaration, semicolon }              |
      "TYPE", { type declaration, semicolon }                   |
      "VAR", { class variable declaration, semicolon }          |
      abstract method declarations , semicolon ;

normal method declarations =
      normal method declaration | overriding method declaration;
normal method declaration = procedure declaration;
overriding method declaration = "OVERRIDE", procedure declaration;

abstract method declarations =
      normal method declaration | abstract method definition | overriding method declaration;

traced class declaration =
      ( normal traced class declaration | abstract traced class declaration ) ;

normal traced class declaration =
      normal traced class header, ( normal traced class declaration body | "FORWARD" ) ;
normal traced class header=
      "TRACED", "CLASS", class identifier, semicolon ;
```

```
normal traced class declaration body =
      [ inherit clause ],
      [ reveal list ],
      normal class component declarations,
      [ traced class body ],
      "END", class identifier ;

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

traced class body = "BEGIN", block body;
```

A.3 Reveal List

```
reveal list = "REVEAL" revealed component list, semicolon ;

revealed component list = revealed component, { comma, revealed component } ;
revealed component = identifier | "READONLY" class variable identifier ;
class variable identifier = identifier ;
```

A.4 Inherit Clause

```
inherit clause = "INHERIT", class type identifier, semicolon ;
class type identifier = type identifier ;
```

A.5 Designators

```
object selected designator =
      object variable designator, period, [ class identifier, period ], class variable identifier ;
object variable designator = variable designator ;

object selected value =
      object value designator, period, [ class identifier, period ], entity identifier ;
object value designator = value designator ;
entity identifier = identifier ;
```

A.6 Guard Statement

```
guard statement =
      "GUARD", guard selector, "AS", guarded list,
      ["ELSE" statement sequence],
      "END";

guard selector = expression ;

guarded list = guarded statement sequence {vertical bar, guarded statement sequence} ;
guarded statement sequence =
      [[object denoter], colon, guarded class type, "DO", statement sequence] ;
guarded class type = class type identifier ;
object denoter = identifier ;
```

Annex B
(informative)

Changes to the Syntax of the Base Language

program module =
["UNSAFEGUARDED"], "MODULE", module identifier,
[interrupt protection], semicolon,
import lists,
module block, module identifier, period ;

definition module =
["UNSAFEGUARDED"], "DEFINITION", "MODULE", module identifier, semicolon,
import lists, definitions,
"END", module identifier, period ;

implementation module =
["UNSAFEGUARDED"], "IMPLEMENTATION", "MODULE", module identifier,
[interrupt protection], semicolon,
import lists,
module block, module identifier, period ;

definition =
"CONST", {constant declaration, semicolon} |
"TYPE", {type definition, semicolon} |
"VAR", {variable declaration, semicolon} |
procedure heading, semicolon |
class definition, semicolon ;

declaration =
"CONST", {constant declaration, semicolon} |
"TYPE", {type declaration, semicolon} |
"VAR", {variable declaration, semicolon} |
procedure declaration, semicolon |
class declaration, semicolon, |
local module declaration, semicolon;

qualified identifier = { qualifying identifier, period }, **[class identifier, period],**
identifier ;

variable designator = entire designator | indexed designator | selected designator |
dereferenced designator | **object selected designator** ;

value designator = entire value | indexed value | selected value | dereferenced value |
object selected value ;

statement =
empty statement | assignment statement | procedure call |
retry statement | with statement | if statement |
case statement | while statement | repeat statement |
loop statement | exit statement | for statement |
guard statement ;

Annex C
(informative)

Glossary

Ancestor
A class that other classes have inherited from. An ancestor class may be the "immediate" or "direct" ancestor (also called the parent class), that is the class that is being directly inherited from. Alternately an ancestor class may be any class further up the inheritance chain that is inherited by a class from which the current class inherits. (see Inheritance)

Attribute
A class variable (i. e. data component of a class).

Child
A class that directly inherits another class, that is the class that lists another class in its own inherit clause. (see Inheritance)

Class
An interface (description) of the data and methods that an object of this class contains. A class also contains the specifications of the data and methods. (In a safe object oriented language the definition and specification can be separated.) A class is a language entity that encapsulates both the data and the operations on the data. A class is essentially an Abstract Data Type (ADT).

Class type
The new type introduced by a class declaration or class definition is called class type.

Client
Any part of a program that uses another part of that program. For example a class that sends a message to an object of another class, or a class that has an object within it of another class is a client of the class to which the object belongs.

Component
Any entity declared inside a class is called component of this class (e. g. class variables, methods).

Descendant
Any class that inherits from another class, including classes further down the inheritance chain that inherit from classes that have inherited from a class. A "direct" or "immediate" descendant (also called a child class) is the class that directly inherits from another class. (see Inheritance)

Dynamic Binding
The ability of a variable to refer to different versions of an object. A variable declared to be of a superclass type can refer to a subclass object, as the subclass object contains all features that the superclass contains. A variable of the superclass type can only access those features that are declared within the superclass, as they are all that the superclass is aware of, even though the object the variable is referring to may contain other features as well.