

# INTERNATIONAL STANDARD

# ISO/IEC 8652

Second edition  
1995-02-15

**AMENDMENT 1**  
2007-03-15

---

---

## Information technology — Programming languages — Ada

### AMENDMENT 1

*Technologies de l'information — Langages de programmation — Ada*  
*AMENDEMENT 1*



Reference number  
ISO/IEC 8652:1995/Amd.1:2007(E)

**AVA** Documents  
Document Sharing Hub  
© ISO/IEC 2007

**PDF disclaimer**

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2007

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office  
Case postale 56 • CH-1211 Geneva 20  
Tel. + 41 22 749 01 11  
Fax + 41 22 749 09 47  
E-mail [copyright@iso.org](mailto:copyright@iso.org)  
Web [www.iso.org](http://www.iso.org)

Published in Switzerland

## 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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. 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.

Amendment 1 to ISO/IEC 8652:1995 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Amendment 1 cancels and replaces those portions of ISO/IEC 8652:1995 as corrected by Technical Corrigendum 1 ISO/IEC 8652:1995/Cor.1:2001 as specified by the body of this amendment. Those portions of the International Standard as corrected by Technical Corrigendum 1 not modified by this amendment remain in force.

The main emphasis of Amendment 1 is to improve the object-oriented programming and the real-time features of ISO/IEC 8652:1995 while also strengthening reliability.

As in ISO/IEC 8652:1995, Annexes A to J form an integral part of the International Standard as amended. Annexes K to Q are for information only.

## Introduction

International Standard ISO/IEC 8652:1995 defines the Ada programming language.

This amendment modifies Ada by making changes and additions that improve:

- The safety of applications written in Ada;
- The portability of applications written in Ada;
- Interoperability with other languages and systems; and
- Accessibility and ease of transition from idioms in other programming and modeling languages.

This amendment incorporates the following major additions to the International Standard:

- Support for the entire ISO/IEC 10646:2003 character repertoire, both in program text and executing programs (see subclauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4);
- Interfaces, to provide a limited form of multiple inheritance of operations (see 3.9.4);
- Improvements for access types, such as null excluding subtypes (see 3.10), additional uses for anonymous access types (see 3.6 and 8.5.1), and anonymous access-to-subprogram subtypes to support “downward closures” (see 3.10 and 3.10.2);
- Additional context clause capabilities: limited views to allow mutually dependent types (see 3.10.1 and 10.1.2) and private with clauses that apply only in the private part of a package (see 10.1.2);
- Aggregates, constructor functions and constants for limited types (see 4.3.1, 6.5, and 7.5);
- Control of overriding to eliminate errors (see 8.3);
- Additional standard packages, including time management (see 9.6), file directory and name management (see A.16), containers (see A.18), execution-time clocks (see D.14), timing events (see D.15), and vector and matrix operations (see G.3);
- A mechanism for writing C unions to make interfaces with C systems easier (see B.3.3);
- New task dispatching policies, including non-preemptive (see D.2.4) and earliest deadline first (see D.2.6); and
- The Ravenscar profile to provide a simplified tasking system for high-integrity systems (see D.13).

This amendment is organized by sections corresponding to those in the International Standard. These sections include wording changes and additions to the International Standard. Clause and subclause headings are given for each clause that contains a wording change. Clauses and subclauses that do not contain any change or addition are omitted.

For each change, an *anchor* paragraph from the International Standard (as corrected by Technical Corrigendum 1) is given. New or revised text and instructions are given with each change. The anchor paragraph can be replaced or deleted, or text can be inserted before or after it. When a heading immediately precedes the anchor paragraph, any text inserted before the paragraph is intended to appear under the heading.

Typographical conventions:

**Instructions about the text changes are in this font.** The actual text changes are in the same fonts as the International Standard – this font for text, this font for syntax, and this font for Ada source code.

# Information technology — Programming languages — Ada

## AMENDMENT 1

### Introduction

*Of International Standard ISO/IEC 8652:1995. Modifications of this section of that International Standard are found here.*

#### Replace paragraph 3:

- Rationale for the Ada Programming Language -- 1995 edition, which gives an introduction to the new features of Ada, and explains the rationale behind them. Programmers should read this first.

**by:**

- Ada 95 Rationale. This gives an introduction to the new features of Ada incorporated in the 1995 edition of this Standard, and explains the rationale behind them. Programmers unfamiliar with Ada 95 should read this first.
- Ada 2005 Rationale. This gives an introduction to the changes and new features in Ada 2005 (compared with the 1995 edition), and explains the rationale behind them. Programmers should read this rationale before reading this Standard in depth.

#### Replace paragraph 5:

- The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the RM95, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

**by:**

- The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the consolidated Ada Reference Manual, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

#### Replace paragraph 6:

Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. This revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency.

**by:**

Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. The 1995 revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency. This amended version provides further flexibility and adds more standardized packages within the framework provided by the 1995 revision.

**Replace paragraph 32:**

An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types Boolean, Character, and Wide\_Character are predefined.

**by:**

An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types Boolean, Character, Wide\_Character, and Wide\_Wide\_Character are predefined.

**Replace paragraph 34:**

Composite types allow definitions of structured objects with related components. The composite types in the language include arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types. Task and protected types are also forms of composite types. The array types String and Wide\_String are predefined.

**by:**

Composite types allow definitions of structured objects with related components. The composite types in the language include arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types. Task and protected types are also forms of composite types. The array types String, Wide\_String, and Wide\_Wide\_String are predefined.

**Insert after paragraph 38:**

From any type a new type may be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations may be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives may be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension must be tagged, so that the specific type of an operand within a derivation class can be identified at run time. When an operation of a tagged type is applied to an operand whose specific type is not known until run time, implicit dispatching is performed based on the tag of the operand.

**the new paragraph:**

Interface types provide abstract models from which other interfaces and types may be composed and derived. This provides a reliable form of multiple inheritance. Interface types may also be implemented by task types and protected types thereby enabling concurrent programming and inheritance to be merged.

**Replace paragraph 41:**

Representation clauses can be used to specify the mapping between types and features of an underlying machine. For example, the user can specify that objects of a given type must be represented with a given number of bits, or that the components of a record are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code.

**by:**

Aspect clauses can be used to specify the mapping between types and features of an underlying machine. For example, the user can specify that objects of a given type must be represented with a given number of bits, or that the components of a record are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code.

**Replace paragraph 42:**

The predefined environment of the language provides for input-output and other capabilities (such as string manipulation and random number generation) by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided. Other standard library packages are defined in annexes of the standard to support systems with specialized requirements.

**by:**

The predefined environment of the language provides for input-output and other capabilities by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

The predefined standard library packages provide facilities such as string manipulation, containers of various kinds (vectors, lists, maps, etc.), mathematical functions, random number generation, and access to the execution environment.

The specialized annexes define further predefined library packages and facilities with emphasis on areas such as real-time scheduling, interrupt handling, distributed systems, numerical computation, and high-integrity systems.

**Replace paragraph 44:**

This International Standard replaces the first edition of 1987. In this edition, the following major language changes have been incorporated:

**by:**

This amended International Standard updates the edition of 1995 which replaced the first edition of 1987. In the 1995 edition, the following major language changes were incorporated.

**Replace paragraph 45:**

- Support for standard 8-bit and 16-bit character sets. See Section 2, 3.5.2, 3.6.3, A.1, A.3, and A.4.

**by:**

- Support for standard 8-bit and 16-bit characters was added. See clauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4.

**Replace paragraph 46:**

- Object-oriented programming with run-time polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses 3.4, 3.9, and 7.3. See also the new forms of generic formal parameters that are allowed by 12.5.1, "Formal Private and Derived Types" and 12.7, "Formal Packages".

**by:**

- The type model was extended to include facilities for object-oriented programming with dynamic polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses 3.4, 3.9, and 7.3. Additional forms of generic formal parameters were allowed as described in clauses 12.5.1 and 12.7.

**Replace paragraph 47:**

- Access types have been extended to allow an access value to designate a subprogram or an object declared by an object declaration (as opposed to just a heap-allocated object). See 3.10.

**by:**

- Access types were extended to allow an access value to designate a subprogram or an object declared by an object declaration as opposed to just an object allocated on a heap. See clause 3.10.

**Replace paragraph 48:**

- Efficient data-oriented synchronization is provided via protected types. See Section 9.

**by:**

- Efficient data-oriented synchronization was provided by the introduction of protected types. See clause 9.4.

**Replace paragraph 49:**

- The library units of a library may be organized into a hierarchy of parent and child units. See Section 10.

**by:**

- The library structure was extended to allow library units to be organized into a hierarchy of parent and child units. See clause 10.1.

**Replace paragraph 50:**

- Additional support has been added for interfacing to other languages. See Annex B.

**by:**

- Additional support was added for interfacing to other languages. See Annex B.

**Replace paragraph 51:**

- The Specialized Needs Annexes have been added to provide specific support for certain application areas:

**by:**

- The Specialized Needs Annexes were added to provide specific support for certain application areas:

**Replace paragraph 57:**

- Annex H, "Safety and Security"

**by:**

- Annex H, "High Integrity Systems"

Amendment 1 modifies the 1995 International Standard by making changes and additions that improve the capability of the language and the reliability of programs written in the language. In particular, the changes were designed to improve the portability of programs, interfacing to other languages, and both the object-oriented and real-time capabilities.

The following significant changes with respect to the 1995 edition are incorporated:

- Support for program text is extended to cover the entire ISO/IEC 10646:2003 repertoire. Execution support now includes the 32-bit character set. See clauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4.
- The object-oriented model has been improved by the addition of an interface facility which provides multiple inheritance and additional flexibility for type extensions. See clauses 3.4, 3.9, and 7.3. An alternative notation for calling operations more akin to that used in other languages has also been added. See clause 4.1.3.
- Access types have been further extended to unify properties such as the ability to access constants and to exclude null values. See clause 3.10. Anonymous access types are now permitted more freely and anonymous access-to-subprogram types are introduced. See clauses 3.3, 3.6, 3.10, and 8.5.1.
- The control of structure and visibility has been enhanced to permit mutually dependent references between units and finer control over access from the private part of a package. See clauses 3.10.1 and 10.1.2. In addition, limited types have been made more useful by the provision of aggregates, constants, and constructor functions. See clauses 4.3, 6.5, and 7.5.
- The predefined environment has been extended to include additional time and calendar operations, improved string handling, a comprehensive container library, file and directory management, and access to environment variables. See clauses 9.6.1, A.4, A.16, A.17, and A.18.
- Two of the Specialized Needs Annexes have been considerably enhanced:
  - The Real-Time Systems Annex now includes the Ravenscar profile for high-integrity systems, further dispatching policies such as Round Robin and Earliest Deadline First, support for timing events, and support for control of CPU time utilization. See clauses D.2, D.13, D.14, and D.15.
  - The Numerics Annex now includes support for real and complex vectors and matrices as previously defined in ISO/IEC 13813:1998 plus further basic operations for linear algebra. See clause G.3.
- The overall reliability of the language has been enhanced by a number of improvements. These include new syntax which detects accidental overloading, as well as pragmas for making assertions and giving better control over the suppression of checks. See clauses 6.1, 11.4.2, and 11.5.

## Section 1: General

### 1.1.2 Structure

Replace paragraph 13:

- Annex H, "Safety and Security"

by:

- Annex H, "High Integrity Systems"

### 1.1.4 Method of Description and Syntax Notation

Replace paragraph 9:

`return_statement ::= return [expression];`

`return_statement ::= return; | return expression;`

by:

`simple_return_statement ::= return [expression];`

`simple_return_statement ::= return; | return expression;`

Insert after paragraph 14:

- If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example *subtype\_name* and *task\_name* are both equivalent to *name* alone.

the new paragraph:

The delimiters, compound delimiters, reserved words, and `numeric_literals` are exclusively made of the characters whose code position is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this International Standard (see 2.1) belong to the same range. For example, the character E in the definition of exponent is the character whose name is "LATIN CAPITAL LETTER E", not "GREEK CAPITAL LETTER EPSILON".

Insert before paragraph 15:

A *syntactic category* is a nonterminal in the grammar defined in BNF under "Syntax." Names of syntactic categories are set in a different font, like *this*.

the new paragraph:

When this International Standard mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using locale-independent full case folding, as defined by documents referenced in the note in section 1 of ISO/IEC 10646:2003.

## 1.2 Normative References

Replace paragraph 3:

ISO/IEC 1539:1991, *Information technology — Programming languages — FORTRAN*.

by:

ISO/IEC 1539-1:2004, *Information technology — Programming languages — Fortran — Part 1: Base language*.

Replace paragraph 4:

ISO 1989:1985, *Programming languages — COBOL*.

by:

ISO/IEC 1989:2002, *Information technology — Programming languages — COBOL*.

**Insert after paragraph 5:**

ISO/IEC 6429:1992, *Information technology — Control functions for coded graphic character sets*.

**the new paragraph:**

ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

**Replace paragraph 7:**

ISO/IEC 9899:1990, *Programming languages — C*.

by:

ISO/IEC 9899:1999, *Programming languages — C*, supplemented by Technical Corrigendum 1:2001 and Technical Corrigendum 2:2004.

**Replace paragraph 8:**

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*, supplemented by Technical Corrigendum 1:1996.

by:

ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

ISO/IEC 14882:2003, *Programming languages — C++*.

ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*.

## 1.3 Definitions

**Replace paragraph 1:**

Terms are defined throughout this International Standard, indicated by *italic* type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to the *Webster's Third New International Dictionary of the English Language*. Informal descriptions of some terms are also given in Annex N, "Glossary".

by:

Terms are defined throughout this International Standard, indicated by *italic* type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Mathematical terms not defined in this International Standard are to be interpreted according to the *CRC Concise Encyclopedia of Mathematics, Second Edition*. Other terms not defined in this International Standard are to be interpreted according to the *Webster's Third New International Dictionary of the English Language*. Informal descriptions of some terms are also given in Annex N, "Glossary".

## Section 2: Lexical Elements

### 2.1 Character Set

#### Replace paragraph 1:

The only characters allowed outside of comments are the `graphic_characters` and `format_effectors`.

by:

The character repertoire for the text of an Ada program consists of the entire coding space described by the ISO/IEC 10646:2003 Universal Multiple-Octet Coded Character Set. This coding space is organized in *planes*, each plane comprising 65536 characters.

#### Delete paragraph 2:

`character ::= graphic_character | format_effector | other_control_function`

#### Replace paragraph 3:

`graphic_character ::= identifier_letter | digit | space_character | special_character`

by:

A **character** is defined by this International Standard for each cell in the coding space described by ISO/IEC 10646:2003, regardless of whether or not ISO/IEC 10646:2003 allocates a character to that cell.

#### Replace paragraph 4:

The character repertoire for the text of an Ada program consists of the collection of characters called the Basic Multilingual Plane (BMP) of the ISO 10646 Universal Multiple-Octet Coded Character Set, plus a set of `format_effectors` and, in comments only, a set of `other_control_functions`; the coded representation for these characters is implementation defined (it need not be a representation defined within ISO-10646-1).

by:

The coded representation for characters is implementation defined (it need not be a representation defined within ISO/IEC 10646:2003). A character whose relative code position in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program.

The semantics of an Ada program whose text is not in Normalization Form KC (as defined by section 24 of ISO/IEC 10646:2003) is implementation defined.

#### Replace paragraph 5:

The description of the language definition in this International Standard uses the graphic symbols defined for Row 00: Basic Latin and Row 00: Latin-1 Supplement of the ISO 10646 BMP; these correspond to the graphic symbols of ISO 8859-1 (Latin-1); no graphic symbols are used in this International Standard for characters outside of Row 00 of the BMP. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

by:

The description of the language definition in this International Standard uses the character properties General Category, Simple Uppercase Mapping, Uppercase Mapping, and Special Case Condition of the documents referenced by the note in section 1 of ISO/IEC 10646:2003. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

#### Replace paragraph 6:

The categories of characters are defined as follows:

by:

Characters are categorized as follows:

#### Delete paragraph 7:

`identifier_letter`  
`upper_case_identifier_letter | lower_case_identifier_letter`

**Replace paragraph 8:**

upper\_case\_identifier\_letter

Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Capital Letter".

**by:**

letter\_uppercase

Any character whose General Category is defined to be "Letter, Uppercase".

**Replace paragraph 9:**

lower\_case\_identifier\_letter

Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Small Letter".

**by:**

letter\_lowercase

Any character whose General Category is defined to be "Letter, Lowercase".

letter\_titlecase

Any character whose General Category is defined to be "Letter, Titlecase".

letter\_modifier

Any character whose General Category is defined to be "Letter, Modifier".

letter\_other

Any character whose General Category is defined to be "Letter, Other".

mark\_non\_spacing

Any character whose General Category is defined to be "Mark, Non-Spacing".

mark\_spacing\_combining

Any character whose General Category is defined to be "Mark, Spacing Combining".

**Replace paragraph 10:**

digit

One of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

**by:**

number\_decimal

Any character whose General Category is defined to be "Number, Decimal".

number\_letter

Any character whose General Category is defined to be "Number, Letter".

punctuation\_connector

Any character whose General Category is defined to be "Punctuation, Connector".

other\_format

Any character whose General Category is defined to be "Other, Format".

**Replace paragraph 11:**

space\_character

The character of ISO 10646 BMP named "Space".

**by:**

separator\_space

Any character whose General Category is defined to be "Separator, Space".

**Replace paragraph 12:**

special\_character

Any character of the ISO 10646 BMP that is not reserved for a control function, and is not the space\_character, an identifier\_letter, or a digit.

**by:**

**separator\_line**

Any character whose General Category is defined to be "Separator, Line".

**separator\_paragraph**

Any character whose General Category is defined to be "Separator, Paragraph".

**Replace paragraph 13:**

**format\_effector**

The control functions of ISO 6429 called character tabulation (HT), line tabulation (VT), carriage return (CR), line feed (LF), and form feed (FF).

**by:**

**format\_effector**

The characters whose code positions are 16#09# (CHARACTER TABULATION), 16#0A# (LINE FEED), 16#0B# (LINE TABULATION), 16#0C# (FORM FEED), 16#0D# (CARRIAGE RETURN), 16#85# (NEXT LINE), and the characters in categories **separator\_line** and **separator\_paragraph**.

**other\_control**

Any character whose General Category is defined to be "Other, Control", and which is not defined to be a **format\_effector**.

**other\_private\_use**

Any character whose General Category is defined to be "Other, Private Use".

**other\_surrogate**

Any character whose General Category is defined to be "Other, Surrogate".

**Replace paragraph 14:**

**other\_control\_function**

Any control function, other than a **format\_effector**, that is allowed in a comment; the set of **other\_control\_functions** allowed in comments is implementation defined.

**by:**

**graphic\_character**

Any character that is not in the categories **other\_control**, **other\_private\_use**, **other\_surrogate**, **format\_effector**, and whose relative code position in its plane is neither 16#FFFE# nor 16#FFFF#.

**Replace paragraph 15:**

The following names are used when referring to certain **special\_characters**:

**by:**

The following names are used when referring to certain characters (the first name is that given in ISO/IEC 10646:2003):

**Delete paragraph 16:**

In a nonstandard mode, the implementation may support a different character repertoire; in particular, the set of characters that are considered **identifier\_letters** can be extended or changed to conform to local conventions.

**Replace paragraph 17:**

1 Every code position of ISO 10646 BMP that is not reserved for a control function is defined to be a **graphic\_character** by this International Standard. This includes all code positions other than 0000 - 001F, 007F - 009F, and FFFE - FFFF.

**by:**

1 The characters in categories **other\_control**, **other\_private\_use**, and **other\_surrogate** are only allowed in comments.

## 2.2 Lexical Elements, Separators, and Delimiters

### Replace paragraph 2:

The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more `format_effectors` other than character tabulation (HT) signifies at least one end of line.

by:

The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more `format_effectors` other than the character whose code position is 16#09# (CHARACTER TABULATION) signifies at least one end of line.

### Replace paragraph 3:

In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a space character, a format effector, or the end of a line, as follows:

by:

In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a `separator_space`, a `format_effector`, or the end of a line, as follows:

### Replace paragraph 4:

- A space character is a separator except within a `comment`, a `string_literal`, or a `character_literal`.

by:

- A `separator_space` is a separator except within a `comment`, a `string_literal`, or a `character_literal`.

### Replace paragraph 5:

- Character tabulation (HT) is a separator except within a `comment`.

by:

- The character whose code position is 16#09# (CHARACTER TABULATION) is a separator except within a `comment`.

### Replace paragraph 8:

A *delimiter* is either one of the following special characters

by:

A *delimiter* is either one of the following characters:

## 2.3 Identifiers

### Replace paragraph 2:

```
identifier ::=
    identifier_letter {[underline] letter_or_digit}
```

by:

```
identifier ::=
    identifier_start {identifier_start | identifier_extend}
```

### Replace paragraph 3:

```
letter_or_digit ::= identifier_letter | digit
```

by:

```

identifier_start ::=
    letter_uppercase
  | letter_lowercase
  | letter_titlecase
  | letter_modifier
  | letter_other
  | number_letter

identifier_extend ::=
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector
  | other_format

```

#### Replace paragraph 4:

An identifier shall not be a reserved word.

by:

After eliminating the characters in category **other\_format**, an identifier shall not contain two consecutive characters in category **punctuation\_connector**, or end with a character in that category.

#### Replace paragraph 5:

All characters of an identifier are significant, including any underline character. Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

by:

Two identifiers are considered the same if they consist of the same sequence of characters after applying the following transformations (in this order):

- The characters in category **other\_format** are eliminated.
- The remaining sequence of characters is converted to upper case.

#### Replace paragraph 6:

In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions.

by:

After applying these transformations, an identifier shall not be identical to a reserved word (in upper case).

In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions.

#### NOTES

- Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

#### Replace paragraph 8:

```

Count      X   Get_Symbol   Ethelyn   Marion
Snobol_4   X1  Page_Count   Store_Next_Item

```

by:

```

Count      X   Get_Symbol   Ethelyn   Marion
Snobol_4   X1  Page_Count   Store_Next_Item
Πλάτων     -- Plato
Чайковский -- Tchaikovsky
Θ φ        -- Angles

```

### 2.4.1 Decimal Literals

**Insert after paragraph 5:**

$$\text{exponent} ::= E [+]\text{ numeral} \mid E - \text{numeral}$$

**the new paragraph:**

$$\text{digit} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

## 2.5 Character Literals

**Replace paragraph 5:**

'A'                '\* '                ' ' '                ' '

**by:**

|     |      |     |                                      |
|-----|------|-----|--------------------------------------|
| 'A' | '*   | ' ' | ' '                                  |
| 'L' | 'II' | 'Λ' | -- Various els.                      |
| '∞' | 'ℵ'  |     | -- Big numbers - infinity and aleph. |

## 2.6 String Literals

**Insert after paragraph 7:**

## NOTES

6 An end of line cannot appear in a `string_literal`.

**the new paragraph:**

7 No transformation is performed on the sequence of characters of a `string_literal`.

**Replace paragraph 9:**

```
"Message of the day:"
""          -- a null string literal
" " "A" ""  -- three string literals of length 1

"Characters such as $, %, and } are allowed in string literals"
```

**by:**

```
"Message of the day:"

""                -- a null string literal
" "      "A"      """"    -- three string literals of length 1

"Characters such as $, %, and } are allowed in string literals"
"Archimedes said ""Εύρηκα""""
"Volume of cylinder (pr²h) = "
```

## 2.8 Pragmas

**Replace paragraph 29:**

```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Inline(Set_Mask); -- generate code for Set_Mask inline
pragma Suppress(Range Check, On => Index); -- turn off range checking on Index
```

**by:**

```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Inline(Set_Mask); -- generate code for Set_Mask inline
pragma Import(C, Put Char, External Name => "putchar"); -- import C putchar function
```

## 2.9 Reserved Words

In paragraph 2 replace:

The following are the *reserved words* (ignoring upper/lower case distinctions):

by:

The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case, and one or more characters in category **other\_format** may be inserted within or at the end of the reserved word.

In the list in paragraph 2, add:

**interface**

**overriding**

**synchronized**

## Section 3: Declarations and Types

### 3.1 Declarations

Replace paragraph 3:

```
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | package_declaration | renaming_declaration
  | exception_declaration | generic_declaration
  | generic_instantiation
```

by:

```
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | null_procedure_declaration | package_declaration
  | renaming_declaration | exception_declaration
  | generic_declaration | generic_instantiation
```

Replace paragraph 6:

Each of the following is defined to be a declaration: any *basic\_declaration*; an *enumeration\_literal\_specification*; a *discriminant\_specification*; a *component\_declaration*; a *loop\_parameter\_specification*; a *parameter\_specification*; a *subprogram\_body*; an *entry\_declaration*; an *entry\_index\_specification*; a *choice\_parameter\_specification*; a *generic\_formal\_parameter\_declaration*.

by:

Each of the following is defined to be a declaration: any *basic\_declaration*; an *enumeration\_literal\_specification*; a *discriminant\_specification*; a *component\_declaration*; a *loop\_parameter\_specification*; a *parameter\_specification*; a *subprogram\_body*; an *entry\_declaration*; an *entry\_index\_specification*; a *choice\_parameter\_specification*; a *generic\_formal\_parameter\_declaration*. In addition, an *extended\_return\_statement* is a declaration of its defining identifier.

### 3.2 Types and Subtypes

Replace paragraph 2:

Types are grouped into *classes* of types, reflecting the similarity of their values and primitive operations. There exist several *language-defined classes* of types (see NOTES below). *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

by:

Types are grouped into *categories* of types. There exist several *language-defined categories* of types (see NOTES below), reflecting the similarity of their values and primitive operations. Most categories of types form *classes* of types. *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

Replace paragraph 4:

The composite types are the *record* types, *record extensions*, *array* types, *task* types, and *protected* types. A *private* type or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. A partial view is a composite type.

by:

The composite types are the *record* types, *record extensions*, *array* types, *interface* types, *task* types, and *protected* types.

There can be multiple views of a type with varying sets of operations. An *incomplete* type represents an incomplete view (see 3.10.1) of a type with a very restricted usage, providing support for recursive data structures. A *private* type or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. The full view (see 3.2.1) of a type represents its complete definition. An incomplete or partial view is considered a composite type, even if the full view is not.

#### Replace paragraph 5:

Certain composite types (and partial views thereof) have special components called *discriminants* whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type.

by:

Certain composite types (and views thereof) have special components called *discriminants* whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type.

#### Replace paragraph 6:

The term *subcomponent* is used in this International Standard in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents.

by:

The term *subcomponent* is used in this International Standard in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents. The terms component, subcomponent, and part are also applied to a type meaning the component, subcomponent, or part of objects and values of the type.

#### Replace paragraph 7:

The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case of a *null constraint* that specifies no restriction is also included); the rules for which values satisfy a given kind of constraint are given in 3.5 for *range\_constraints*, 3.6.1 for *index\_constraints*, and 3.7.1 for *discriminant\_constraints*.

by:

The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case of a *null constraint* that specifies no restriction is also included); the rules for which values satisfy a given kind of constraint are given in 3.5 for *range\_constraints*, 3.6.1 for *index\_constraints*, and 3.7.1 for *discriminant\_constraints*. The set of possible values for an object of an access type can also be subjected to a condition that excludes the null value (see 3.10).

#### Replace paragraph 8:

A *subtype* of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype. The given type is called the *type of the subtype*. Similarly, the associated constraint is called the *constraint of the subtype*. The set of values of a subtype consists of the values of its type that satisfy its constraint. Such values *belong* to the subtype.

by:

A *subtype* of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype. The given type is called the *type of the subtype*. Similarly, the associated constraint is called the *constraint of the subtype*. The set of values of a subtype consists of the values of its type that satisfy its constraint and any exclusion of the null value. Such values *belong* to the subtype.

#### Replace paragraph 10:

2 Any set of types that is closed under derivation (see 3.4) can be called a "class" of types. However, only certain classes are used in the description of the rules of the language —generally those that have their own particular set of primitive operations (see 3.2.3), or that correspond to a set of types that are matched by a given kind of generic formal

type (see 12.5). The following are examples of "interesting" *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric, access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes.

by:

2 Any set of types can be called a "category" of types, and any set of types that is closed under derivation (see 3.4) can be called a "class" of types. However, only certain categories and classes are used in the description of the rules of the language —generally those that have their own particular set of primitive operations (see 3.2.3), or that correspond to a set of types that are matched by a given kind of generic formal type (see 12.5). The following are examples of "interesting" *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric, access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes. In addition to these classes, the following are examples of "interesting" *language-defined categories*: abstract, incomplete, interface, limited, private, record.

**Replace paragraph 11:**

These language-defined classes are organized like this:

by:

These language-defined categories are organized like this:

**Replace paragraph 12:**

- all types
  - elementary
  - scalar
  - discrete
    - enumeration
      - character
      - boolean
      - other enumeration
  - integer
    - signed integer
    - modular integer
  - real
    - floating point
    - fixed point
      - ordinary fixed point
      - decimal fixed point
- access
  - access-to-object
  - access-to-subprogram
- composite
  - array
    - string
    - other array
- untagged record
- tagged
- task
- protected

**by:**

- all types
  - elementary
    - scalar
      - discrete
        - enumeration
          - character
          - boolean
          - other enumeration
        - integer
          - signed integer
          - modular integer
      - real
        - floating point
        - fixed point
          - ordinary fixed point
          - decimal fixed point
  - access
    - access-to-object
    - access-to-subprogram
  - composite
    - untagged
      - array
        - string
        - other array
    - record
    - task
      - protected
    - tagged (including interfaces)
      - nonlimited tagged record
      - limited tagged
        - limited tagged record
      - synchronized tagged
        - tagged task
        - tagged protected

**Replace paragraph 13:**

The classes "numeric" and "nonlimited" represent other classification dimensions and do not fit into the above strictly hierarchical picture.

**by:**

There are other categories, such as "numeric" and "discriminated", which represent other categorization dimensions, but do not fit into the above strictly hierarchical picture.

### 3.2.1 Type Declarations

#### Replace paragraph 4:

```
type_definition ::=
  enumeration_type_definition | integer_type_definition
  | real_type_definition | array_type_definition
  | record_type_definition | access_type_definition
  | derived_type_definition
```

#### by:

```
type_definition ::=
  enumeration_type_definition | integer_type_definition
  | real_type_definition | array_type_definition
  | record_type_definition | access_type_definition
  | derived_type_definition | interface_type_definition
```

#### Replace paragraph 7:

A type defined by a **type\_declaration** is a *named* type; such a type has one or more nameable subtypes. Certain other forms of declaration also include type definitions as part of the declaration for an object (including a parameter or a discriminant). The type defined by such a declaration is *anonymous* — it has no nameable subtypes. For explanatory purposes, this International Standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an **identifier**. For a named type whose first subtype is T, this International Standard sometimes refers to the type of T as simply "the type T."

#### by:

A type defined by a **type\_declaration** is a *named* type; such a type has one or more nameable subtypes. Certain other forms of declaration also include type definitions as part of the declaration for an object. The type defined by such a declaration is *anonymous* — it has no nameable subtypes. For explanatory purposes, this International Standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an **identifier**. For a named type whose first subtype is T, this International Standard sometimes refers to the type of T as simply "the type T".

#### Replace paragraph 8:

A named type that is declared by a **full\_type\_declaration**, or an anonymous type that is defined as part of declaring an object of the type, is called a *full type*. The **type\_definition**, **task\_definition**, **protected\_definition**, or **access\_definition** that defines a full type is called a *full type definition*. Types declared by other forms of **type\_declaration** are not separate types; they are partial or incomplete views of some full type.

#### by:

A named type that is declared by a **full\_type\_declaration**, or an anonymous type that is defined by an **access\_definition** or as part of declaring an object of the type, is called a *full type*. The declaration of a full type also declares the *full view* of the type. The **type\_definition**, **task\_definition**, **protected\_definition**, or **access\_definition** that defines a full type is called a *full type definition*. Types declared by other forms of **type\_declaration** are not separate types; they are partial or incomplete views of some full type.

### 3.2.2 Subtype Declarations

#### Replace paragraph 3:

```
subtype_indication ::= subtype_mark [constraint]
```

#### by:

```
subtype_indication ::= [null_exclusion] subtype_mark [constraint]
```

**Replace paragraph 15:**

```

subtype Rainbow    is Color range Red .. Blue;           -- see 3.2.1
subtype Red_Blue   is Rainbow;
subtype Int         is Integer;
subtype Small_Int   is Integer range -10 .. 10;
subtype Up_To_K     is Column range 1 .. K;                 -- see 3.2.1
subtype Square      is Matrix(1 .. 10, 1 .. 10);             -- see 3.6
subtype Male        is Person(Sex => M);                     -- see 3.10.1

```

**by:**

```

subtype Rainbow    is Color range Red .. Blue;           -- see 3.2.1
subtype Red_Blue   is Rainbow;
subtype Int         is Integer;
subtype Small_Int   is Integer range -10 .. 10;
subtype Up_To_K     is Column range 1 .. K;                 -- see 3.2.1
subtype Square      is Matrix(1 .. 10, 1 .. 10);             -- see 3.6
subtype Male        is Person(Sex => M);                     -- see 3.10.1
subtype Binop_Ref   is not null Binop_Ptr;                   -- see 3.10

```

**3.2.3 Classification of Operations****Replace paragraph 1:**

An operation *operates on a type* *T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter (see 6.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

**by:**

An operation *operates on a type* *T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter or access result type (see 6.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

**Replace paragraph 7:**

- Any subprograms not covered above that are explicitly declared immediately within the same declarative region as the type and that override (see 8.3) other implicitly declared primitive subprograms of the type.

**by:**

- For a nonformal type, any subprograms not covered above that are explicitly declared immediately within the same declarative region as the type and that override (see 8.3) other implicitly declared primitive subprograms of the type.

**3.3 Objects and Named Numbers****Replace paragraph 10:**

- the result of evaluating a `function_call` (or the equivalent operator invocation — see 6.6);

**by:**

- the return object created as the result of evaluating a `function_call` (or the equivalent operator invocation — see 6.6);

**3.3.1 Object Declarations****Replace paragraph 2:**

```
object_declaration ::=
```

```

    defining_identifier_list : [aliased] [constant] subtype_indication [= expression]
| defining_identifier_list : [aliased] [constant] array_type_definition [= expression]
| single_task_declaration
| single_protected_declaration

```

by:

```

object_declaration ::=
    defining_identifier_list : [aliased] [constant] subtype_indication [= expression]
| defining_identifier_list : [aliased] [constant] access_definition [= expression]
| defining_identifier_list : [aliased] [constant] array_type_definition [= expression]
| single_task_declaration
| single_protected_declaration

```

#### Replace paragraph 5:

An `object_declaration` without the reserved word **constant** declares a variable object. If it has a `subtype_indication` or an `array_type_definition` that defines an indefinite subtype, then there shall be an initialization expression. An initialization expression shall not be given if the object is of a limited type.

by:

An `object_declaration` without the reserved word **constant** declares a variable object. If it has a `subtype_indication` or an `array_type_definition` that defines an indefinite subtype, then there shall be an initialization expression.

#### Replace paragraph 8:

The `subtype_indication` or full type definition of an `object_declaration` defines the nominal subtype of the object. The `object_declaration` declares an object of the type of the nominal subtype.

by:

The `subtype_indication`, `access_definition`, or full type definition of an `object_declaration` defines the nominal subtype of the object. The `object_declaration` declares an object of the type of the nominal subtype.

A component of an object is said to *require late initialization* if it has an access discriminant value constrained by a per-object expression, or if it has an initialization expression that includes a name denoting the current instance of the type or denoting an access discriminant.

#### Replace paragraph 9:

If a composite object declared by an `object_declaration` has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant or aliased (see 3.10) the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; the object is said to be *constrained by its initial value*. In the case of an aliased object, this initial value may be either explicit or implicit; in the other cases, an explicit initial value is required. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a *constrained object*.

by:

If a composite object declared by an `object_declaration` has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; the object is said to be *constrained by its initial value*. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a *constrained object*.

#### Replace paragraph 16:

1. The `subtype_indication`, `array_type_definition`, `single_task_declaration`, or `single_protected_declaration` is first elaborated. This creates the nominal subtype (and the anonymous type in the latter three cases).

by:

1. The `subtype_indication`, `access_definition`, `array_type_definition`, `single_task_declaration`, or `single_protected_declaration` is first elaborated. This creates the nominal subtype (and the anonymous type in the last four cases).

**Replace paragraph 18:**

3. The object is created, and, if there is not an initialization expression, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype.

**by:**

3. The object is created, and, if there is not an initialization expression, the object is *initialized by default*. When an object is initialized by default, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype. Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in 5.2 and 7.6, Initialize and Adjust procedures can be called.

**Delete paragraph 19:**

4. Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in 5.2 and 7.6, Initialize and Adjust procedures can be called.

**Replace paragraph 20:**

For the third step above, the object creation and any elaborations and evaluations are performed in an arbitrary order, except that if the **default\_expression** for a discriminant is evaluated to obtain its initial value, then this evaluation is performed before that of the **default\_expression** for any component that depends on the discriminant, and also before that of any **default\_expression** that includes the name of the discriminant. The evaluations of the third step and the assignments of the fourth step are performed in an arbitrary order, except that each evaluation is performed before the resulting value is assigned.

**by:**

For the third step above, evaluations and assignments are performed in an arbitrary order subject to the following restrictions:

- Assignment to any part of the object is preceded by the evaluation of the value that is to be assigned.
- The evaluation of a **default\_expression** that includes the name of a discriminant is preceded by the assignment to that discriminant.
- The evaluation of the **default\_expression** for any component that depends on a discriminant is preceded by the assignment to that discriminant.
- The assignments to any components, including implicit components, not requiring late initialization must precede the initial value evaluations for any components requiring late initialization; if two components both require late initialization, then assignments to parts of the component occurring earlier in the order of the component declarations must precede the initial value evaluations of the component occurring later.

**Replace paragraph 27:**

```
John, Paul : Person_Name := new Person(Sex => M); -- see 3.10.1
```

**by:**

```
John, Paul : not null Person_Name := new Person(Sex => M); -- see 3.10.1
```

**Replace paragraph 29:**

```
John : Person_Name := new Person(Sex => M);
Paul : Person_Name := new Person(Sex => M);
```

**by:**

```
John : not null Person_Name := new Person(Sex => M);
Paul : not null Person_Name := new Person(Sex => M);
```

**Replace paragraph 31:**

```
Count, Sum : Integer;
Size       : Integer range 0 .. 10_000 := 0;
Sorted     : Boolean := False;
Color_Table : array(1 .. Max) of Color;
```

```
Option      : Bit_Vector(1 .. 10) := (others => True);
Hello      : constant String := "Hi, world.";
```

by:

```
Count, Sum  : Integer;
Size       : Integer range 0 .. 10_000 := 0;
Sorted     : Boolean := False;
Color_Table : array(1 .. Max) of Color;
Option     : Bit_Vector(1 .. 10) := (others => True);
Hello      : aliased String := "Hi, world.";
θ, φ       : Float range -π .. +π;
```

Replace paragraph 33:

```
Limit      : constant Integer := 10_000;
Low_Limit  : constant Integer := Limit/10;
Tolerance  : constant Real := Dispersion(1.15);
```

by:

```
Limit      : constant Integer := 10_000;
Low_Limit  : constant Integer := Limit/10;
Tolerance  : constant Real := Dispersion(1.15);
Hello_Msg  : constant access String := Hello'Access; -- see 3.10.2
```

### 3.3.2 Number Declarations

Replace paragraph 10:

```
Max          : constant := 500;                -- an integer number
Max_Line_Size : constant := Max/6              -- the integer 83
Power_16     : constant := 2**16;             -- the integer 65_536
One, Un, Eins : constant := 1;                -- three different names for 1
```

by:

```
Max          : constant := 500;                -- an integer number
Max_Line_Size : constant := Max/6              -- the integer 83
Power_16     : constant := 2**16;             -- the integer 65_536
One, Un, Eins : constant := 1;                -- three different names for 1
```

### 3.4 Derived Types and Classes

Replace paragraph 1:

A `derived_type_definition` defines a new type (and its first subtype) whose characteristics are *derived* from those of a *parent type*.

by:

A `derived_type_definition` defines a *derived type* (and its first subtype) whose characteristics are *derived* from those of a parent type, and possibly from progenitor types.

A *class of types* is a set of types that is closed under derivation; that is, if the parent or a progenitor type of a derived type belongs to a class, then so does the derived type. By saying that a particular group of types forms a class, we are saying that all derivatives of a type in the set inherit the characteristics that define that set. The more general term *category of types* is used for a set of types whose defining characteristics are not necessarily inherited by derivatives; for example, limited, abstract, and interface are all categories of types, but not classes of types.

Replace paragraph 2:

```
derived_type_definition ::= [abstract] new parent_subtype_indication [record_extension_part]
```

by:

```
derived_type_definition ::=
  [abstract] [limited] new parent_subtype_indication [[and interface_list] record_extension_part]
```

**Replace paragraph 3:**

The *parent\_subtype\_indication* defines the *parent subtype*; its type is the parent type.

**by:**

The *parent\_subtype\_indication* defines the *parent subtype*; its type is the *parent type*. The *interface\_list* defines the progenitor types (see 3.9.4). A derived type has one parent type and zero or more progenitor types.

**Replace paragraph 5:**

If there is a *record\_extension\_part*, the derived type is called a *record extension* of the parent type. A *record\_extension\_part* shall be provided if and only if the parent type is a tagged type.

**by:**

If there is a *record\_extension\_part*, the derived type is called a *record extension* of the parent type. A *record\_extension\_part* shall be provided if and only if the parent type is a tagged type. An *interface\_list* shall be provided only if the parent type is a tagged type.

If the reserved word **limited** appears in a *derived\_type\_definition*, the parent type shall be a limited type.

**Insert after paragraph 6:**

The first subtype of the derived type is unconstrained if a *known\_discriminant\_part* is provided in the declaration of the derived type, or if the parent subtype is unconstrained. Otherwise, the constraint of the first subtype *corresponds* to that of the parent subtype in the following sense: it is the same as that of the parent subtype except that for a range constraint (implicit or explicit), the value of each bound of its range is replaced by the corresponding value of the derived type.

**the new paragraph:**

The first subtype of the derived type excludes null (see 3.10) if and only if the parent subtype excludes null.

**Replace paragraph 8:**

- Each class of types that includes the parent type also includes the derived type.

**by:**

- If the parent type or a progenitor type belongs to a class of types, then the derived type also belongs to that class. The following sets of types, as well as any higher-level sets composed from them, are classes in this sense, and hence the characteristics defining these classes are inherited by derived types from their parent or progenitor types: signed integer, modular integer, ordinary fixed, decimal fixed, floating point, enumeration, boolean, character, access-to-constant, general access-to-variable, pool-specific access-to-variable, access-to-subprogram, array, string, non-array composite, nonlimited, untagged record, tagged, task, protected, and synchronized tagged.

**Delete paragraph 15:**

- The derived type is limited if and only if the parent type is limited.

**Replace paragraph 17:**

- For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type that already exists at the place of the *derived\_type\_definition*, there exists a corresponding *inherited* primitive subprogram of the derived type with the same defining name. Primitive user-defined equality operators of the parent type are also inherited by the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is rather incorporated into the implementation of the predefined equality operator of the record extension (see 4.5.2).

by:

- For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type or of a progenitor type that already exists at the place of the **derived\_type\_definition**, there exists a corresponding *inherited* primitive subprogram of the derived type with the same defining name. Primitive user-defined equality operators of the parent type and any progenitor types are also inherited by the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is rather incorporated into the implementation of the predefined equality operator of the record extension (see 4.5.2).

**Replace paragraph 18:**

The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent type with a *corresponding subtype* of the derived type. For a given subtype of the parent type, the corresponding subtype of the derived type is defined as follows:

by:

The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent or progenitor type with a *corresponding subtype* of the derived type. For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

**Replace paragraph 22:**

The same formal parameters have **default\_expressions** in the profile of the inherited subprogram. Any type mismatch due to the systematic replacement of the parent type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.

by:

The same formal parameters have **default\_expressions** in the profile of the inherited subprogram. Any type mismatch due to the systematic replacement of the parent or progenitor type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.

**Replace paragraph 23:**

If a primitive subprogram of the parent type is visible at the place of the **derived\_type\_definition**, then the corresponding inherited subprogram is implicitly declared immediately after the **derived\_type\_definition**. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in 7.3.1.

by:

If a primitive subprogram of the parent or progenitor type is visible at the place of the **derived\_type\_definition**, then the corresponding inherited subprogram is implicitly declared immediately after the **derived\_type\_definition**. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in 7.3.1.

**Replace paragraph 27:**

For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see 6.4.1) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the parent's subprogram is converted to the derived type.

by:

For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent or progenitor type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see 6.4.1) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the subprogram of the parent or progenitor is converted to the derived type, or in the case of a null extension, extended to the derived type using the equivalent of an **extension\_aggregate** with the original result as the **ancestor\_part** and **null record** as the **record\_component\_association\_list**.

**Insert after paragraph 35:**

17 If the reserved word **abstract** is given in the declaration of a type, the type is abstract (see 3.9.3).

**the new paragraphs:**

18 An interface type that has a progenitor type "is derived from" that type. A `derived_type_definition`, however, never defines an interface type.

19 It is illegal for the parent type of a `derived_type_definition` to be a synchronized tagged type.

**3.4.1 Derivation Classes****Replace paragraph 2:**

A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. The derivation class of types for a type *T* (also called the class *rooted* at *T*) is the set consisting of *T* (the *root type* of the class) and all types derived from *T* (directly or indirectly) plus any associated universal or class-wide types (defined below).

**by:**

A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. A derived type, interface type, type extension, task type, protected type, or formal derived type is also derived from every ancestor of each of its progenitor types, if any. The derivation class of types for a type *T* (also called the class *rooted* at *T*) is the set consisting of *T* (the *root type* of the class) and all types derived from *T* (directly or indirectly) plus any associated universal or class-wide types (defined below).

**Replace paragraph 3:**

Every type is either a *specific* type, a *class-wide* type, or a *universal* type. A specific type is one defined by a `type_declaration`, a `formal_type_declaration`, or a full type definition embedded in a declaration for an object. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

**by:**

Every type is either a *specific* type, a *class-wide* type, or a *universal* type. A specific type is one defined by a `type_declaration`, a `formal_type_declaration`, or a full type definition embedded in another construct. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

**Replace paragraph 6:**

## Universal types

Universal types are defined for (and belong to) the integer, real, and fixed point classes, and are referred to in this standard as respectively, *universal\_integer*, *universal\_real*, and *universal\_fixed*. These are analogous to class-wide types for these language-defined numeric classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real `numeric_literal`) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6).

**by:**

## Universal types

Universal types are defined for (and belong to) the integer, real, fixed point, and access classes, and are referred to in this standard as respectively, *universal\_integer*, *universal\_real*, *universal\_fixed*, and *universal\_access*. These are analogous to class-wide types for these language-defined elementary classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real `numeric_literal`, or the literal `null`) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6).

**Replace paragraph 10:**

A specific type *T2* is defined to be a *descendant* of a type *T1* if *T2* is the same as *T1*, or if *T2* is derived (directly or indirectly) from *T1*. A class-wide type *T2*'Class is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the universal types are defined to be descendants of the root types of their classes. If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. The *ultimate ancestor* of a type is the ancestor of the type that is not a descendant of any other type.

**by:**

A specific type *T2* is defined to be a *descendant* of a type *T1* if *T2* is the same as *T1*, or if *T2* is derived (directly or indirectly) from *T1*. A class-wide type *T2*'Class is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the numeric universal types are defined to be descendants of the root types of their classes. If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. An *ultimate ancestor* of a type is an ancestor of that type that is not itself a descendant of any other type. Every untagged type has a unique ultimate ancestor.

**3.5 Scalar Types****Insert after paragraph 27:**

For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; *Constraint\_Error* is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of *Arg*. For a fixed point type, the function returns the result of subtracting small from the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of *Arg*; *Constraint\_Error* is raised if there is no such machine number.

**the new paragraphs:**

S'Wide\_Wide\_Image

S'Wide\_Wide\_Image denotes a function with the following specification:

```
function S'Wide_Wide_Image(Arg : S'Base)
return Wide_Wide_String
```

The function returns an *image* of the value of *Arg*, that is, a sequence of characters representing the value in display form. The lower bound of the result is one.

The image of an integer value is the corresponding decimal literal, without underlines, leading zeros, exponent, or trailing spaces, but with a single leading character that is either a minus sign or a space.

The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. For a *nongraphic character* (a value of a character type that has no enumeration literal associated with it), the result is a corresponding language-defined name in upper case (for example, the image of the nongraphic character identified as *nul* is "NUL" — the quotes are not part of the image).

The image of a floating point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, S'Digits–1 (see 3.5.8) digits after the decimal point (but one if S'Digits is one), an upper case E, the sign of the exponent (either + or –), and two or more digits (with leading zeros if necessary) representing the exponent. If S'Signed\_Zeros is True, then the leading character is a minus sign for a negatively signed zero.

The image of a fixed point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and S'Aft (see 3.5.10) digits after the decimal point.

**Replace paragraph 30:**

The function returns an *image* of the value of *Arg*, that is, a sequence of characters representing the value in display form. The lower bound of the result is one.

by:

The function returns an image of the value of *Arg* as a *Wide\_String*. The lower bound of the result is one. The image has the same sequence of character as defined for *S'Wide\_Wide\_Image* if all the graphic characters are defined in *Wide\_Character*; otherwise the sequence of characters is implementation defined (but no shorter than that of *S'Wide\_Wide\_Image* for the same value of *Arg*).

#### Delete paragraph 31:

The image of an integer value is the corresponding decimal literal, without underlines, leading zeros, exponent, or trailing spaces, but with a single leading character that is either a minus sign or a space.

#### Delete paragraph 32:

The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. For a *nongraphic character* (a value of a character type that has no enumeration literal associated with it), the result is a corresponding language-defined or implementation-defined name in upper case (for example, the image of the nongraphic character identified as *nul* is "NUL" — the quotes are not part of the image).

#### Delete paragraph 33:

The image of a floating point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, *S'Digits-1* (see 3.5.8) digits after the decimal point (but one if *S'Digits* is one), an upper case E, the sign of the exponent (either + or –), and two or more digits (with leading zeros if necessary) representing the exponent. If *S'Signed\_Zeros* is True, then the leading character is a minus sign for a negatively signed zero.

#### Delete paragraph 34:

The image of a fixed point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and *S'Aft* (see 3.5.10) digits after the decimal point.

#### Replace paragraph 37:

The function returns an image of the value of *Arg* as a *String*. The lower bound of the result is one. The image has the same sequence of graphic characters as that defined for *S'Wide\_Image* if all the graphic characters are defined in *Character*; otherwise the sequence of characters is implementation defined (but no shorter than that of *S'Wide\_Image* for the same value of *Arg*).

by:

The function returns an image of the value of *Arg* as a *String*. The lower bound of the result is one. The image has the same sequence of graphic characters as that defined for *S'Wide\_Wide\_Image* if all the graphic characters are defined in *Character*; otherwise the sequence of characters is implementation defined (but no shorter than that of *S'Wide\_Wide\_Image* for the same value of *Arg*).

*S'Wide\_Wide\_Width*

*S'Wide\_Wide\_Width* denotes the maximum length of a *Wide\_Wide\_String* returned by *S'Wide\_Wide\_Image* over all values of the subtype *S*. It denotes zero for a subtype that has a null range. Its type is *universal\_integer*.

#### Insert after paragraph 39:

*S'Width*

*S'Width* denotes the maximum length of a *String* returned by *S'Image* over all values of the subtype *S*. It denotes zero for a subtype that has a null range. Its type is *universal\_integer*.

**the new paragraphs:**

S'Wide\_Wide\_Value

S'Wide\_Wide\_Value denotes a function with the following specification:

```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)
return S'Base
```

This function returns a value given an image of the value as a Wide\_Wide\_String, ignoring any leading or trailing spaces.

For the evaluation of a call on S'Wide\_Wide\_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide\_Wide\_Image for a nongraphic character of the type), the result is the corresponding enumeration value; otherwise Constraint\_Error is raised.

For the evaluation of a call on S'Wide\_Wide\_Value for an integer subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint\_Error is raised.

For the evaluation of a call on S'Wide\_Wide\_Value for a real subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following:

- numeric\_literal
- numeral.[exponent]
- .numeral[exponent]
- base#based\_numeral.#[exponent]
- base#.based\_numeral#[exponent]

with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint\_Error is raised. The sign of a zero value is preserved (positive if none has been specified) if S'Signed\_Zeros is True.

**Replace paragraph 43:**

For the evaluation of a call on S'Wide\_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide\_Image for a nongraphic character of the type), the result is the corresponding enumeration value; otherwise Constraint\_Error is raised.

**by:**

For the evaluation of a call on S'Wide\_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide\_Image for a value of the type), the result is the corresponding enumeration value; otherwise Constraint\_Error is raised. For a numeric subtype S, the evaluation of a call on S'Wide\_Value with Arg of type Wide\_String is equivalent to a call on S'Wide\_Wide\_Value for a corresponding Arg of type Wide\_Wide\_String.

**Delete paragraph 44:**

For the evaluation of a call on S'Wide\_Value (or S'Value) for an integer subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint\_Error is raised.

**Delete paragraph 45:**

For the evaluation of a call on S'Wide\_Value (or S'Value) for a real subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following:

**Delete paragraph 46:**

- numeric\_literal

**Delete paragraph 47:**

- numeral.[exponent]

**Delete paragraph 48:**

- .numeral[exponent]

**Delete paragraph 49:**

- base#based\_numeral.#[exponent]

**Delete paragraph 50:**

- base#.based\_numeral#[exponent]

**Delete paragraph 51:**

with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise Constraint\_Error is raised. The sign of a zero value is preserved (positive if none has been specified) if S'Signed\_Zeros is True.

**Replace paragraph 55:**

For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type), the result is the corresponding enumeration value; otherwise Constraint\_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with Arg of type String is equivalent to a call on S'Wide\_Value for a corresponding Arg of type Wide\_String.

**by:**

For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type), the result is the corresponding enumeration value; otherwise Constraint\_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with Arg of type String is equivalent to a call on S'Wide\_Wide\_Value for a corresponding Arg of type Wide\_Wide\_String.

**Replace paragraph 56:**

An implementation may extend the Wide\_Value, Value, Wide\_Image, and Image attributes of a floating point type to support special values such as infinities and NaNs.

**by:**

An implementation may extend the Wide\_Wide\_Value, Wide\_Value, Value, Wide\_Wide\_Image, Wide\_Image, and Image attributes of a floating point type to support special values such as infinities and NaNs.

**Replace paragraph 59:**

21 For any value V (including any nongraphic character) of an enumeration subtype S, S'Value(S'Image(V)) equals V, as does S'Wide\_Value(S'Wide\_Image(V)). Neither expression ever raises Constraint\_Error.

**by:**

21 For any value V (including any nongraphic character) of an enumeration subtype S, S'Value(S'Image(V)) equals V, as do S'Wide\_Value(S'Wide\_Image(V)) and S'Wide\_Wide\_Value(S'Wide\_Wide\_Image(V)). None of these expressions ever raise Constraint\_Error.

### 3.5.2 Character Types

#### Replace paragraph 2:

The predefined type `Character` is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO 10646 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding `character_literal` in `Character`. Each of the nongraphic positions of Row 00 (0000-001F and 007F-009F) has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes `(Wide_)Image` and `(Wide_)Value`; these names are given in the definition of type `Character` in A.1, "The Package Standard", but are set in *italics*.

#### by:

The predefined type `Character` is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding `character_literal` in `Character`. Each of the nongraphic positions of Row 00 (0000-001F and 007F-009F) has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes `Image`, `Wide_Image`, `Wide_Wide_Image`, `Value`, `Wide_Value`, and `Wide_Wide_Value`; these names are given in the definition of type `Character` in A.1, "The Package Standard", but are set in *italics*.

#### Replace paragraph 3:

The predefined type `Wide_Character` is a character type whose values correspond to the 65536 code positions of the ISO 10646 Basic Multilingual Plane (BMP). Each of the graphic characters of the BMP has a corresponding `character_literal` in `Wide_Character`. The first 256 values of `Wide_Character` have the same `character_literal` or language-defined name as defined for `Character`. The last 2 values of `Wide_Character` correspond to the nongraphic positions FFFE and FFFF of the BMP, and are assigned the language-defined names *FFFE* and *FFFF*. As with the other language-defined names for nongraphic characters, the names *FFFE* and *FFFF* are usable only with the attributes `(Wide_)Image` and `(Wide_)Value`; they are not usable as enumeration literals. All other values of `Wide_Character` are considered graphic characters, and have a corresponding `character_literal`.

#### by:

The predefined type `Wide_Character` is a character type whose values correspond to the 65536 code positions of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of the BMP has a corresponding `character_literal` in `Wide_Character`. The first 256 values of `Wide_Character` have the same `character_literal` or language-defined name as defined for `Character`. Each of the `graphic_characters` has a corresponding `character_literal`.

The predefined type `Wide_Wide_Character` is a character type whose values correspond to the 2147483648 code positions of the ISO/IEC 10646:2003 character set. Each of the `graphic_characters` has a corresponding `character_literal` in `Wide_Wide_Character`. The first 65536 values of `Wide_Wide_Character` have the same `character_literal` or language-defined name as defined for `Wide_Character`.

The characters whose code position is larger than 16#FF# and which are not `graphic_characters` have language-defined names which are formed by appending to the string "Hex\_" the representation of their code position in hexadecimal as eight extended digits. As with other language-defined names, these names are usable only with the attributes `(Wide_)Wide_Image` and `(Wide_)Wide_Value`; they are not usable as enumeration literals.

#### Delete paragraph 4:

In a nonstandard mode, an implementation may provide other interpretations for the predefined types `Character` and `Wide_Character`, to conform to local conventions.

#### Delete paragraph 5:

If an implementation supports a mode with alternative interpretations for `Character` and `Wide_Character`, the set of graphic characters of `Character` should nevertheless remain a proper subset of the set of graphic characters of `Wide_Character`. Any character set "localizations" should be reflected in the results of the subprograms defined in the language-defined package `Characters.Handling` (see A.3) available in such a mode. In a mode with an alternative interpretation of `Character`, the implementation should also support a corresponding change in what is a legal `identifier_letter`.

### 3.5.4 Integer Types

#### Replace paragraph 16:

For every modular subtype S, the following attribute is defined:

by:

For every modular subtype S, the following attributes are defined:

S'Mod

S'Mod denotes a function with the following specification:

```
function S'Mod (Arg : universal_integer)
return S'Base
```

This function returns *Arg mod S'Modulus*, as a value of the type of S.

### 3.5.9 Fixed Point Types

#### Replace paragraph 8:

The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. For a type defined by an *ordinary\_fixed\_point\_definition* (an *ordinary* fixed point type), the *small* may be specified by an *attribute\_definition\_clause* (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*.

by:

The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. The *machine numbers* of a fixed point type are the values of the type that can be represented exactly in every unconstrained variable of the type. For a type defined by an *ordinary\_fixed\_point\_definition* (an *ordinary* fixed point type), the *small* may be specified by an *attribute\_definition\_clause* (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*.

### 3.6 Array Types

#### Replace paragraph 7:

component\_definition ::= [aliased] subtype\_indication

by:

```
component_definition ::=
    [aliased] subtype_indication
  | [aliased] access_definition
```

#### Delete paragraph 11:

Within the definition of a nonlimited composite type (or a limited composite type that later in its immediate scope becomes nonlimited — see 7.3.1 and 7.5), if a *component\_definition* contains the reserved word **aliased** and the type of the component is discriminated, then the nominal subtype of the component shall be constrained.

#### Replace paragraph 22:

The elaboration of a *discrete\_subtype\_definition* that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the *subtype\_indication* or the evaluation of the *range*. The elaboration of a *discrete\_subtype\_definition* that contains one or more per-object expressions is defined in 3.8. The elaboration of a *component\_definition* in an *array\_type\_definition* consists of the elaboration of the *subtype\_indication*. The elaboration of any *discrete\_subtype\_definitions* and the elaboration of the *component\_definition* are performed in an arbitrary order.

by:

The elaboration of a `discrete_subtype_definition` that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the `subtype_indication` or the evaluation of the `range`. The elaboration of a `discrete_subtype_definition` that contains one or more per-object expressions is defined in 3.8. The elaboration of a `component_definition` in an `array_type_definition` consists of the elaboration of the `subtype_indication` or `access_definition`. The elaboration of any `discrete_subtype_definitions` and the elaboration of the `component_definition` are performed in an arbitrary order.

Replace paragraph 30:

```
Grid : array(1 .. 80, 1 .. 100) of Boolean;
Mix  : array(Color range Red .. Green) of Boolean;
Page : array(Positive range <>) of Line := -- an array of arrays
      (1 | 50 => Line'(1 | Line'Last => '+', others => '-'), -- see 4.3.3
       2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
      -- Page is constrained by its initial value to (1..50)
```

by:

```
Grid      : array(1 .. 80, 1 .. 100) of Boolean;
Mix       : array(Color range Red .. Green) of Boolean;
Msg_Table : constant array(Error_Code) of access constant String :=
      (Too_Big => new String("Result too big"), Too_Small => ...);
Page      : array(Positive range <>) of Line := -- an array of arrays
      (1 | 50 => Line'(1 | Line'Last => '+', others => '-'), -- see 4.3.3
       2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
      -- Page is constrained by its initial value to (1..50)
```

## 3.6.2 Operations of Array Types

Replace paragraph 16:

48 A component of an array can be named with an `indexed_component`. A value of an array type can be specified with an `array_aggregate`, unless the array type is limited. For a one-dimensional array type, a slice of the array can be named; also, string literals are defined if the component type is a character type.

by:

48 A component of an array can be named with an `indexed_component`. A value of an array type can be specified with an `array_aggregate`. For a one-dimensional array type, a slice of the array can be named; also, string literals are defined if the component type is a character type.

## 3.6.3 String Types

Replace paragraph 2:

There are two predefined string types, `String` and `Wide_String`, each indexed by values of the predefined subtype `Positive`; these are declared in the visible part of package `Standard`:

by:

There are three predefined string types, `String`, `Wide_String`, and `Wide_Wide_String`, each indexed by values of the predefined subtype `Positive`; these are declared in the visible part of package `Standard`:

Replace paragraph 4:

```
type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
```

by:

```
type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;
```

### 3.7 Discriminants

#### Replace paragraph 1:

A composite type (other than an array type) can have discriminants, which parameterize the type. A `known_discriminant_part` specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An `unknown_discriminant_part` in the declaration of a partial view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a partial view are indefinite subtypes.

#### by:

A composite type (other than an array or interface type) can have discriminants, which parameterize the type. A `known_discriminant_part` specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An `unknown_discriminant_part` in the declaration of a view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a view are indefinite subtypes.

#### Replace paragraph 5:

```
discriminant_specification ::=
    defining_identifier_list : subtype_mark [:= default_expression]
    | defining_identifier_list : access_definition [:= default_expression]
```

#### by:

```
discriminant_specification ::=
    defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
    | defining_identifier_list : access_definition [:= default_expression]
```

#### Replace paragraph 8:

A `discriminant_part` is only permitted in a declaration for a composite type that is not an array type (this includes generic formal types). A type declared with a `known_discriminant_part` is called a *discriminated* type, as is a type that inherits (known) discriminants.

#### by:

A `discriminant_part` is only permitted in a declaration for a composite type that is not an array or interface type (this includes generic formal types). A type declared with a `known_discriminant_part` is called a *discriminated* type, as is a type that inherits (known) discriminants.

#### Replace paragraph 9:

The subtype of a discriminant may be defined by a `subtype_mark`, in which case the `subtype_mark` shall denote a discrete or access subtype, or it may be defined by an `access_definition` (in which case the `subtype_mark` of the `access_definition` may denote any kind of subtype). A discriminant that is defined by an `access_definition` is called an *access discriminant* and is of an anonymous general access-to-variable type whose designated subtype is denoted by the `subtype_mark` of the `access_definition`.

#### by:

The subtype of a discriminant may be defined by an optional `null_exclusion` and a `subtype_mark`, in which case the `subtype_mark` shall denote a discrete or access subtype, or it may be defined by an `access_definition`. A discriminant that is defined by an `access_definition` is called an *access discriminant* and is of an anonymous access type.

#### Replace paragraph 10:

A `discriminant_specification` for an access discriminant shall appear only in the declaration for a task or protected type, or for a type with the reserved word **limited** in its (full) definition or in that of one of its ancestors. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

by:

Default\_expressions shall be provided either for all or for none of the discriminants of a known\_discriminant\_part. No default\_expressions are permitted in a known\_discriminant\_part in a declaration of a tagged type or a generic formal type.

A discriminant\_specification for an access discriminant may have a default\_expression only in the declaration for a task or protected type, or for a type that is a descendant of an explicitly limited record type. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

#### Delete paragraph 11:

Default\_expressions shall be provided either for all or for none of the discriminants of a known\_discriminant\_part. No default\_expressions are permitted in a known\_discriminant\_part in a declaration of a tagged type [or a generic formal type].

#### Replace paragraph 27:

An access\_definition is elaborated when the value of a corresponding access discriminant is defined, either by evaluation of its default\_expression or by elaboration of a discriminant\_constraint. The elaboration of an access\_definition creates the anonymous access type. When the expression defining the access discriminant is evaluated, it is converted to this anonymous access type (see 4.6).

by:

For an access discriminant, its access\_definition is elaborated when the value of the access discriminant is defined: by evaluation of its default\_expression, by elaboration of a discriminant\_constraint, or by an assignment that initializes the enclosing object.

#### Replace paragraph 37:

```
type Item(Number : Positive) is
  record
    Content : Integer;
    -- no component depends on the discriminant
  end record;
```

by:

```
task type Worker(Prio : System.Priority; Buf : access Buffer) is
  -- discriminants used to parameterize the task type (see 9.1)
  pragma Priority(Prio); -- see D.1
  entry Fill;
  entry Drain;
end Worker;
```

### 3.7.1 Discriminant Constraints

#### Replace paragraph 7:

A discriminant\_constraint is only allowed in a subtype\_indication whose subtype\_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of a general access subtype, a discriminant\_constraint is illegal if there is a place within the immediate scope of the designated subtype where the designated subtype's view is constrained.

by:

A discriminant\_constraint is only allowed in a subtype\_indication whose subtype\_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a discriminant\_constraint is illegal if the designated type has a partial view that is constrained or, for a general access subtype, has default\_expressions for its discriminants. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. In a generic body, this rule is checked presuming all formal access types of the generic might be general access types, and all untagged discriminated formal types of the generic might have default\_expressions for their discriminants.

### 3.8 Record Types

#### Delete paragraph 8:

A default\_expression is not permitted if the component is of a limited type.

#### Replace paragraph 9:

Each component\_declaration declares a *component* of the record type. Besides components declared by component\_declarations, the components of a record type include any components declared by discriminant\_specifications of the record type declaration. The identifiers of all components of a record type shall be distinct.

#### by:

Each component\_declaration declares a component of the record type. Besides components declared by component\_declarations, the components of a record type include any components declared by discriminant\_specifications of the record type declaration. The identifiers of all components of a record type shall be distinct.

#### Insert before paragraph 14:

The component\_definition of a component\_declaration defines the (nominal) subtype of the component. If the reserved word **aliased** appears in the component\_definition, then the component is aliased (see 3.10).

#### the new paragraph:

If a record\_type\_declaration includes the reserved word **limited**, the type is called an *explicitly limited record type*.

#### Replace paragraph 18:

Within the definition of a composite type, if a component\_definition or discrete\_subtype\_definition (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute\_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. For the elaboration of a component\_definition of a component\_declaration or the discrete\_subtype\_definition of an entry\_declaration for an entry family (see 9.5.2), if the constraint or range of the subtype\_indication or discrete\_subtype\_definition is not a per-object constraint, then the subtype\_indication or discrete\_subtype\_definition is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant.

#### by:

Within the definition of a composite type, if a component\_definition or discrete\_subtype\_definition (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute\_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. For the elaboration of a component\_definition of a component\_declaration or the discrete\_subtype\_definition of an entry\_declaration for an entry family (see 9.5.2), if the component subtype is defined by an access\_definition or if the constraint or range of the subtype\_indication or discrete\_subtype\_definition is not a per-object constraint, then the access\_definition, subtype\_indication, or discrete\_subtype\_definition is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant.

#### Replace paragraph 25:

61 A component of a record can be named with a selected\_component. A value of a record can be specified with a record\_aggregate, unless the record type is limited.

#### by:

61 A component of a record can be named with a selected\_component. A value of a record can be specified with a record\_aggregate.

### 3.9 Tagged Types and Type Extensions

#### Replace paragraph 2:

A record type or private type that has the reserved word **tagged** in its declaration is called a *tagged* type. When deriving from a tagged type, additional components may be defined. As for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden. The derived type is called an *extension* of the ancestor type, or simply a *type extension*. Every type extension is also a tagged type, and is either a *record extension* or a *private extension* of some other tagged type. A record extension is defined by a *derived\_type\_definition* with a *record\_extension\_part*. A private extension, which is a partial view of a record extension, can be declared in the visible part of a package (see 7.3) or in a generic formal part (see 12.5.1).

#### by:

A record type or private type that has the reserved word **tagged** in its declaration is called a *tagged* type. In addition, an interface type is a tagged type, as is a task or protected type derived from an interface (see 3.9.4). When deriving from a tagged type, as for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden. The derived type is called an *extension* of its ancestor types, or simply a *type extension*.

Every type extension is also a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a non-interface synchronized tagged type (see 3.9.4). A record extension is defined by a *derived\_type\_definition* with a *record\_extension\_part* (see 3.9.1), which may include the definition of additional components. A private extension, which is a partial view of a record extension or of a synchronized tagged type, can be declared in the visible part of a package (see 7.3) or in a generic formal part (see 12.5.1).

#### Replace paragraph 4:

The tag of a specific tagged type identifies the *full\_type\_declaration* of the type. If a declaration for a tagged type occurs within a *generic\_package\_declaration*, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body, the language does not specify whether repeated instantiations of the generic body result in distinct tags.

#### by:

The tag of a specific tagged type identifies the *full\_type\_declaration* of the type, and for a type extension, is sufficient to uniquely identify the type among all descendants of the same ancestor. If a declaration for a tagged type occurs within a *generic\_package\_declaration*, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body and with all of its ancestors (if any) also local to the generic body, the language does not specify whether repeated instantiations of the generic body result in distinct tags.

#### Replace paragraph 6:

```
package Ada.Tags is
  type Tag is private;
```

#### by:

```
package Ada.Tags is
  pragma Preelaborate(Tags);
  type Tag is private;

  No_Tag : constant Tag;
```

#### Insert after paragraph 7:

```
function Expanded_Name(T : Tag) return String;
function External_Tag(T : Tag) return String;
function Internal_Tag(External : String) return Tag;
```

#### the new paragraphs:

```
function Descendant_Tag(External : String; Ancestor : Tag) return Tag;
function Is_Descendant_At_Same_Level(Descendant, Ancestor : Tag)
  return Boolean;

function Parent_Tag (T : Tag) return Tag;

type Tag_Array is array (Positive range <>) of Tag;

function Interface_Ancestor_Tags (T : Tag) return Tag_Array;
```

**Insert after paragraph 9:**

```

private
    ... -- not specified by the language
end Ada.Tags;

```

**the new paragraph:**

No\_Tag is the default initial value of type Tag.

**Replace paragraph 10:**

The function Expanded\_Name returns the full expanded name of the first subtype of the specific type identified by the tag, in upper case, starting with a root library unit. The result is implementation defined if the type is declared within an unnamed **block\_statement**.

**by:**

The function Wide\_Wide\_Expanded\_Name returns the full expanded name of the first subtype of the specific type identified by the tag, in upper case, starting with a root library unit. The result is implementation defined if the type is declared within an unnamed **block\_statement**.

The function Expanded\_Name (respectively, Wide\_Expanded\_Name) returns the same sequence of graphic characters as that defined for Wide\_Wide\_Expanded\_Name, if all the graphic characters are defined in Character (respectively, Wide\_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide\_Wide\_Expanded\_Name for the same value of the argument.

**Insert after paragraph 11:**

The function External\_Tag returns a string to be used in an external representation for the given tag. The call External\_Tag(S'Tag) is equivalent to the **attribute\_reference** S'External\_Tag (see 13.3).

**the new paragraph:**

The string returned by the functions Expanded\_Name, Wide\_Expanded\_Name, Wide\_Wide\_Expanded\_Name, and External\_Tag has lower bound 1.

**Replace paragraph 12:**

The function Internal\_Tag returns the tag that corresponds to the given external tag, or raises Tag\_Error if the given string is not the external tag for any specific type of the partition.

**by:**

The function Internal\_Tag returns a tag that corresponds to the given external tag, or raises Tag\_Error if the given string is not the external tag for any specific type of the partition. Tag\_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created (see 13.14).

The function Descendant\_Tag returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the Ancestor tag and has the same accessibility level as the identified ancestor. Tag\_Error is raised if External is not the external tag for such a type. Tag\_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created.

The function Is\_Descendant\_At\_Same\_Level returns True if the Descendant tag identifies a type that is both a descendant of the type identified by Ancestor and at the same accessibility level. If not, it returns False.

The function Parent\_Tag returns the tag of the parent type of the type whose tag is T. If the type does not have a parent type (that is, it was not declared by a derived\_type\_declaration), then No\_Tag is returned.

The function Interface\_Ancursor\_Tags returns an array containing the tag of each interface ancestor type of the type whose tag is T, other than T itself. The lower bound of the returned array is 1, and the order of the returned tags is unspecified. Each tag appears in the result exactly once. If the type whose tag is T has no interface ancestors, a null array is returned.

**Insert after paragraph 18:**

X'Tag

X'Tag denotes the tag of X. The value of this attribute is of type Tag.

**the new paragraphs:**

The following language-defined generic function exists:

```
generic
  type T (<>) is abstract tagged limited private;
  type Parameters (<>) is limited private;
  with function Constructor (Params : not null access Parameters)
    return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
  (The_Tag : Tag;
   Params : not null access Parameters) return T'Class;
pragma Preelaborate(Generic_Dispatching_Constructor);
pragma Convention(Intrinsic, Generic_Dispatching_Constructor);
```

Tags.Generic\_Dispatching\_Constructor provides a mechanism to create an object of an appropriate type from just a tag value. The function Constructor is expected to create the object given a reference to an object of type Parameters.

**Replace paragraph 24:**

- The tag of the result returned by a function with a class-wide result type is that of the return expression.

by:

- The tag of the result returned by a function with a class-wide result type is that of the return object.

**Insert after paragraph 25:**

The tag is preserved by type conversion and by parameter passing. The tag of a value is the tag of the associated object (see 6.2).

**the new paragraphs:**

Tag\_Error is raised by a call of Descendant\_Tag, Expanded\_Name, External\_Tag, Interface\_Ancestors\_Tag, Is\_Descendant\_At\_Same\_Level, or Parent\_Tag if any tag passed is No\_Tag.

An instance of Tags.Generic\_Dispatching\_Constructor raises Tag\_Error if The\_Tag does not represent a concrete descendant of T or if the innermost master (see 7.6.1) of this descendant is not also a master of the instance. Otherwise, it dispatches to the primitive function denoted by the formal Constructor for the type identified by The\_Tag, passing Params, and returns the result. Any exception raised by the function is propagated.

*Erroneous Execution*

If an internal tag provided to an instance of Tags.Generic\_Dispatching\_Constructor or to any subprogram declared in package Tags identifies either a type that is not library-level and whose tag has not been created (see 13.14), or a type that does not exist in the partition at the time of the call, then execution is erroneous.

**Replace paragraph 26:**

The implementation of the functions in Ada.Tags may raise Tag\_Error if no specific type corresponding to the tag passed as a parameter exists in the partition at the time the function is called.

by:

The implementation of Internal\_Tag and Descendant\_Tag may raise Tag\_Error if no specific type corresponding to the string External passed as a parameter exists in the partition at the time the function is called, or if there is no such type whose innermost master is a master of the point of the function call.

*Implementation Advice*

Internal\_Tag should return the tag of a type whose innermost master is the master of the point of the function call.

**Replace paragraph 30:**

65 If S denotes an untagged private type whose full type is tagged, then S'Class is also allowed before the full type definition, but only in the private part of the package in which the type is declared (see 7.3.1). Similarly, the Class attribute is defined for incomplete types whose full type is tagged, but only within the library unit in which the incomplete type is declared (see 3.10.1).

by:

65 The capability provided by `Tags.Generic_Dispatching_Constructor` is sometimes known as a *factory*.

### 3.9.1 Type Extensions

#### Replace paragraph 1:

Every type extension is a tagged type, and is either a *record extension* or a *private extension* of some other tagged type.

by:

Every type extension is a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a non-interface synchronized tagged type.

#### Replace paragraph 3:

The parent type of a record extension shall not be a class-wide type. If the parent type is nonlimited, then each of the components of the `record_extension_part` shall be nonlimited. The accessibility level (see 3.10.2) of a record extension shall not be statically deeper than that of its parent type. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

by:

The parent type of a record extension shall not be a class-wide type nor shall it be a synchronized tagged type (see 3.9.4). If the parent type or any progenitor is nonlimited, then each of the components of the `record_extension_part` shall be nonlimited. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

#### Replace paragraph 4:

A type extension shall not be declared in a generic body if the parent type is declared outside that body.

by:

Within the body of a generic unit, or the body of any of its descendant library units, a tagged type shall not be declared as a descendant of a formal type declared within the formal part of the generic unit.

*Static Semantics*

A record extension is a *null extension* if its declaration has no `known_discriminant_part` and its `record_extension_part` includes no `component_declarations`.

#### Replace paragraph 7:

The accessibility rules imply that a tagged type declared in a `library_package_specification` can be extended only at library level or as a generic formal. When the extension is declared immediately within a `package_body`, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added.

by:

When an extension is declared immediately within a body, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added.

### 3.9.2 Dispatching Operations of Tagged Types

#### Replace paragraph 1:

The primitive subprograms of a tagged type are called *dispatching operations*. A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time; such a call is termed a *dispatching call*. As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.

by:

The primitive subprograms of a tagged type, the subprograms declared by `formal_abstract_subprogram_declarations`, and the stream attributes of a specific tagged type that are available (see 13.13.2) at the end of the declaration list where the type is declared are called *dispatching operations*. A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time; such a call is termed a *dispatching call*. As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.

#### Replace paragraph 2:

A call on a *dispatching operation* is a call whose **name** or **prefix** denotes the declaration of a primitive subprogram of a tagged type, that is, a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T*, then the call has a *controlling result* — the context of the call can control the dispatching.

by:

A call on a *dispatching operation* is a call whose **name** or **prefix** denotes the declaration of a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T*, then the call has a *controlling result* — the context of the call can control the dispatching. Similarly, if the call is to a function with access result type designating *T*, then the call has a *controlling access result*, and the context can similarly control dispatching.

#### Replace paragraph 4:

- The **name** or expression is *statically tagged* if it is of a specific tagged type and, if it is a call with a controlling result, it has at least one statically tagged controlling operand;

by:

- The **name** or expression is *statically tagged* if it is of a specific tagged type and, if it is a call with a controlling result or controlling access result, it has at least one statically tagged controlling operand;

#### Replace paragraph 5:

- The **name** or expression is *dynamically tagged* if it is of a class-wide type, or it is a call with a controlling result and at least one dynamically tagged controlling operand;

by:

- The **name** or expression is *dynamically tagged* if it is of a class-wide type, or it is a call with a controlling result or controlling access result and at least one dynamically tagged controlling operand;

#### Replace paragraph 6:

- The **name** or expression is *tag indeterminate* if it is a call with a controlling result, all of whose controlling operands (if any) are tag indeterminate.

by:

- The **name** or expression is *tag indeterminate* if it is a call with a controlling result or controlling access result, all of whose controlling operands (if any) are tag indeterminate.

**Replace paragraph 10:**

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited or overriding dispatching operation is the convention of the corresponding primitive operation of the parent type. An explicitly declared dispatching operation shall not be of convention Intrinsic.

**by:**

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited dispatching operation is the convention of the corresponding primitive operation of the parent or progenitor type. The default convention of a dispatching operation that overrides an inherited primitive operation is the convention of the inherited operation; if the operation overrides multiple inherited operations, then they shall all have the same convention. An explicitly declared dispatching operation shall not be of convention Intrinsic.

**Replace paragraph 11:**

The `default_expression` for a controlling formal parameter of a dispatching operation shall be tag indeterminate. A controlling formal parameter that is an access parameter shall not have a `default_expression`.

**by:**

The `default_expression` for a controlling formal parameter of a dispatching operation shall be tag indeterminate.

If a dispatching operation is defined by a `subprogram_renaming_declaration` or the instantiation of a generic subprogram, any access parameter of the renamed subprogram or the generic subprogram that corresponds to a controlling access parameter of the dispatching operation, shall have a subtype that excludes null.

**Replace paragraph 17:**

If all of the controlling operands are tag-indeterminate, then:

**by:**

If all of the controlling operands (if any) are tag-indeterminate, then:

**Replace paragraph 18:**

- If the call has a controlling result and is itself a (possibly parenthesized or qualified) controlling operand of an enclosing call on a dispatching operation of type *T*, then its controlling tag value is determined by the controlling tag value of this enclosing call;

**by:**

- If the call has a controlling result or controlling access result and is itself, or designates, a (possibly parenthesized or qualified) controlling operand of an enclosing call on a dispatching operation of a descendant of type *T*, then its controlling tag value is determined by the controlling tag value of this enclosing call;
- If the call has a controlling result or controlling access result and (possibly parenthesized, qualified, or dereferenced) is the expression of an `assignment_statement` whose target is of a class-wide type, then its controlling tag value is determined by the target;

**Replace paragraph 20:**

For the execution of a call on a dispatching operation, the body executed is the one for the corresponding primitive subprogram of the specific type identified by the controlling tag value. The body for an explicitly declared dispatching operation is the corresponding explicit body for the subprogram. The body for an implicitly declared dispatching operation that is overridden is the body for the overriding subprogram, even if the overriding occurs in a private part. The body for an inherited dispatching operation that is not overridden is the body of the corresponding subprogram of the parent or ancestor type.

by:

For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by the controlling tag value. If the corresponding operation is explicitly declared for this type, even if the declaration occurs in a private part, then the action comprises an invocation of the explicit body for the operation. If the corresponding operation is implicitly declared for this type:

- if the operation is implemented by an entry or protected subprogram (see 9.1 and 9.4), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any;
- otherwise, the action is the same as the action for the corresponding operation of the parent type.

**Replace paragraph 22:**

73 This subclause covers calls on primitive subprograms of a tagged type. Rules for tagged type membership tests are described in 4.5.2. Controlling tag determination for an `assignment_statement` is described in 5.2.

by:

73 This subclause covers calls on dispatching subprograms of a tagged type. Rules for tagged type membership tests are described in 4.5.2. Controlling tag determination for an `assignment_statement` is described in 5.2.

### 3.9.3 Abstract Types and Subprograms

**Replace paragraph 1:**

An *abstract type* is a tagged type intended for use as a parent type for type extensions, but which is not allowed to have objects of its own. An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.

by:

An *abstract type* is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own. An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.

*Syntax*

```
abstract_subprogram_declaration ::=
    [overriding_indicator]
    subprogram_specification is abstract;
```

*Static Semantics*

Interface types (see 3.9.4) are abstract types. In addition, a tagged type that has the reserved word **abstract** in its declaration is an abstract type. The class-wide type (see 3.4.1) rooted at an abstract type is not itself an abstract type.

**Replace paragraph 2:**

An abstract type is a specific type that has the reserved word **abstract** in its declaration. Only a tagged type is allowed to be declared **abstract**.

by:

Only a tagged type shall have the reserved word **abstract** in its declaration.

**Replace paragraph 3:**

A subprogram declared by an `abstract_subprogram_declaration` (see 6.1) is an *abstract subprogram*. If it is a primitive subprogram of a tagged type, then the tagged type shall be **abstract**.

by:

A subprogram declared by an `abstract_subprogram_declaration` or a `formal_abstract_subprogram_declaration` (see 12.6) is an *abstract subprogram*. If it is a primitive subprogram of a tagged type, then the tagged type shall be **abstract**.

**Replace paragraph 4:**

For a derived type, if the parent or ancestor type has an abstract primitive subprogram, or a primitive function with a controlling result, then:

**by:**

If a type has an implicitly declared primitive subprogram that is inherited or is the predefined equality operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a type other than a null extension inherits a function with a controlling result, then:

**Replace paragraph 5:**

- If the derived type is abstract or untagged, the inherited subprogram is *abstract*.

**by:**

- If the type is abstract or untagged, the implicitly declared subprogram is *abstract*.

**Replace paragraph 6:**

Otherwise, the subprogram shall be overridden with a nonabstract subprogram; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

**by:**

Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a function with a controlling result, have a full type that is a null extension; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. Such a subprogram is said to *require overriding*. However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

**Replace paragraph 11:**

A generic actual subprogram shall not be an abstract subprogram. The prefix of an `attribute_reference` for the `Access`, `Unchecked_Access`, or `Address` attributes shall not denote an abstract subprogram.

**by:**

A generic actual subprogram shall not be an abstract subprogram unless the generic formal subprogram is declared by a `formal_abstract_subprogram_declaration`. The prefix of an `attribute_reference` for the `Access`, `Unchecked_Access`, or `Address` attributes shall not denote an abstract subprogram.

*Dynamic Semantics*

The elaboration of an `abstract_subprogram_declaration` has no effect.

### 3.9.4 Interface Types

**Insert new clause:**

An interface type is an abstract tagged type that provides a restricted form of multiple inheritance. A tagged type, task type, or protected type may have one or more interface types as ancestors.

*Syntax*

```
interface_type_definition ::=
  [limited | task | protected | synchronized] interface [and interface_list]
  interface_list ::= interface_subtype_mark {and interface_subtype_mark}
```

*Static Semantics*

An interface type (also called an *interface*) is a specific abstract tagged type that is defined by an `interface_type_definition`.

An interface with the reserved word **limited**, **task**, **protected**, or **synchronized** in its definition is termed, respectively, a *limited interface*, a *task interface*, a *protected interface*, or a *synchronized interface*. In addition, all task and protected interfaces are synchronized interfaces, and all synchronized interfaces are limited interfaces.

A task or protected type derived from an interface is a tagged type. Such a tagged type is called a *synchronized* tagged type, as are synchronized interfaces and private extensions whose declaration includes the reserved word **synchronized**.

A task interface is an abstract task type. A protected interface is an abstract protected type.

An interface type has no components.

An *interface\_subtype\_mark* in an *interface\_list* names a *progenitor subtype*; its type is the *progenitor type*. An interface type inherits user-defined primitive subprograms from each progenitor type in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4).

#### Legality Rules

All user-defined primitive subprograms of an interface type shall be abstract subprograms or null procedures.

The type of a subtype named in an *interface\_list* shall be an interface type.

A type derived from a nonlimited interface shall be nonlimited.

An interface derived from a task interface shall include the reserved word **task** in its definition; any other type derived from a task interface shall be a private extension or a task type declared by a task declaration (see 9.1).

An interface derived from a protected interface shall include the reserved word **protected** in its definition; any other type derived from a protected interface shall be a private extension or a protected type declared by a protected declaration (see 9.4).

An interface derived from a synchronized interface shall include one of the reserved words **task**, **protected**, or **synchronized** in its definition; any other type derived from a synchronized interface shall be a private extension, a task type declared by a task declaration, or a protected type declared by a protected declaration.

No type shall be derived from both a task interface and a protected interface.

In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

#### Dynamic Semantics

The elaboration of an *interface\_type\_definition* has no effect.

#### NOTES

79 Nonlimited interface types have predefined nonabstract equality operators. These may be overridden with user-defined abstract equality operators. Such operators will then require an explicit overriding for any nonabstract descendant of the interface.

#### Examples

*Example of a limited interface and a synchronized interface extending it:*

```

type Queue is limited interface;
procedure Append(Q : in out Queue; Person : in Person_Name) is abstract;
procedure Remove_First(Q : in out Queue;
                        Person : out Person_Name) is abstract;
function Cur_Count(Q : in Queue) return Natural is abstract;
function Max_Count(Q : in Queue) return Natural is abstract;
-- See 3.10.1 for Person_Name.

Queue_Error : exception;
-- Append raises Queue_Error if Count(Q) = Max_Count(Q)
-- Remove_First raises Queue_Error if Count(Q) = 0

type Synchronized_Queue is synchronized interface and Queue; -- see 9.11
procedure Append_Wait(Q : in out Synchronized_Queue;
                      Person : in Person_Name) is abstract;
procedure Remove_First_Wait(Q : in out Synchronized_Queue;
                             Person : out Person_Name) is abstract;

...

procedure Transfer(From : in out Queue'Class;
```

```

        To      : in out Queue'Class;
        Number : in      Natural := 1) is
    Person : Person_Name;
begin
    for I in 1..Number loop
        Remove_First(From, Person);
        Append(To, Person);
    end loop;
end Transfer;

```

This defines a Queue interface defining a queue of people. (A similar design could be created to define any kind of queue simply by replacing Person\_Name by an appropriate type.) The Queue interface has four dispatching operations, Append, Remove\_First, Cur\_Count, and Max\_Count. The body of a class-wide operation, Transfer is also shown. Every non-abstract extension of Queue must provide implementations for at least its four dispatching operations, as they are abstract. Any object of a type derived from Queue may be passed to Transfer as either the From or the To operand. The two operands need not be of the same type in any given call.

The Synchronized\_Queue interface inherits the four dispatching operations from Queue and adds two additional dispatching operations, which wait if necessary rather than raising the Queue\_Error exception. This synchronized interface may only be implemented by a task or protected type, and as such ensures safe concurrent access.

*Example use of the interface:*

```

type Fast_Food_Queue is new Queue with record ...;
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue; Person : in Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;

...

Cashier, Counter : Fast_Food_Queue;

...
-- Add George (see 3.10.1) to the cashier's queue:
Append (Cashier, George);
-- After payment, move George to the sandwich counter queue:
Transfer (Cashier, Counter);
...

```

An interface such as Queue can be used directly as the parent of a new type (as shown here), or can be used as a progenitor when a type is derived. In either case, the primitive operations of the interface are inherited. For Queue, the implementation of the four inherited routines must be provided. Inside the call of Transfer, calls will dispatch to the implementations of Append and Remove\_First for type Fast\_Food\_Queue.

*Example of a task interface:*

```

type Serial_Device is task interface; -- see 9.1
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
procedure Write(Dev : in Serial_Device; C : in Character) is abstract;

```

The Serial\_Device interface has two dispatching operations which are intended to be implemented by task entries (see 9.1).

### 3.10 Access Types

Replace paragraph 2:

```

access_type_definition ::=
    access_to_object_definition
    | access_to_subprogram_definition

```

by:

```

access_type_definition ::=
    [null_exclusion] access_to_object_definition
    | [null_exclusion] access_to_subprogram_definition

```

**Replace paragraph 6:**

`access_definition ::= access subtype_mark`

**by:**

`null_exclusion ::= not null`

`access_definition ::=`

`[null_exclusion] access [constant] subtype_mark`  
`| [null_exclusion] access [protected] procedure parameter_profile`  
`| [null_exclusion] access [protected] function parameter_and_result_profile`

**Replace paragraph 9:**

A view of an object is defined to be *aliased* if it is defined by an `object_declaration` or `component_definition` with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. Finally, the current instance of a limited type, and a formal parameter or generic formal object of a tagged type are defined to be aliased. Aliased views are the ones that can be designated by an access value. If the view defined by an `object_declaration` is aliased, and the type of the object has discriminants, then the object is constrained; if its nominal subtype is unconstrained, then the object is constrained by its initial value. Similarly, if the object created by an `allocator` has discriminants, the object is constrained, either by the designated subtype, or by its initial value.

**by:**

A view of an object is defined to be *aliased* if it is defined by an `object_declaration` or `component_definition` with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. The current instance of a limited tagged type, a protected type, a task type, or a type that has the reserved word **limited** in its full definition is also defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. Aliased views are the ones that can be designated by an access value.

**Replace paragraph 12:**

An `access_definition` defines an anonymous general access-to-variable type; the `subtype_mark` denotes its *designated subtype*. An `access_definition` is used in the specification of an access discriminant (see 3.7) or an access parameter (see 6.1).

**by:**

An `access_definition` defines an anonymous general access type or an anonymous access-to-subprogram type. For a general access type, the `subtype_mark` denotes its *designated subtype*; if the `general_access_modifier constant` appears, the type is an access-to-constant type; otherwise it is an access-to-variable type. For an access-to-subprogram type, the `parameter_profile` or `parameter_and_result_profile` denotes its *designated profile*.

**Replace paragraph 13:**

For each (named) access type, there is a literal **null** which has a null access value designating no entity at all. The null value of a named access type is the default initial value of the type. Other values of an access type are obtained by evaluating an `attribute_reference` for the `Access` or `Unchecked_Access` attribute of an aliased view of an object or non-intrinsic subprogram, or, in the case of a named access-to-object type, an `allocator`, which returns an access value designating a newly created object (see 3.10.2).

**by:**

For each access type, there is a null access value designating no entity at all, which can be obtained by (implicitly) converting the literal **null** to the access type. The null value of an access type is the default initial value of the type. Non-null values of an access-to-object type are obtained by evaluating an `allocator`, which returns an access value designating a newly created object (see 3.10.2), or in the case of a general access-to-object type, evaluating an `attribute_reference` for the `Access` or `Unchecked_Access` attribute of an aliased view of an object. Non-null values of an access-to-subprogram type are obtained by evaluating an `attribute_reference` for the `Access` attribute of a non-intrinsic subprogram.

A `null_exclusion` in a construct specifies that the null value does not belong to the access subtype defined by the construct, that is, the access subtype *excludes null*. In addition, the anonymous access subtype defined by the `access_definition` for a controlling access parameter (see 3.9.2) excludes null. Finally, for a `subtype_indication`