INTERNATIONAL STANDARD

# ISO/IEC 8652

Fourth edition
2023-05

# Information technology — Programming languages — Ada

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

Reference number
ISO/IEC 8652:2023(E)

Documents
Document Sharing Hub
© ISO/IEC 2023

Full document details: ISO_IEC 8652_2023

**COPYRIGHT PROTECTED DOCUMENT**

© ISO/IEC 2023

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland



© ISO/IEC 2023 – All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 8652_2023

# Contents


Foreword ... xiii

Introduction ... xv

1 Scope ... 1

2 Normative references ... 1

3 Terms and definitions ... 2

3.1 Types, objects, and their properties ... 2
3.2 Subprograms and their properties ... 7
3.3 Other syntactic constructs ... 7
3.4 Runtime actions ... 10
3.5 Exceptional situations ... 10

4 General ... 11

4.1 Structure ... 11
4.2 Conformity of an Implementation ... 12
4.3 Method of Description and Syntax Notation ... 14
4.4 Classification of Errors ... 15

5 Lexical Elements ... 16

5.1 Character Set ... 16
5.2 Lexical Elements, Separators, and Delimiters ... 18
5.3 Identifiers ... 19
5.4 Numeric Literals ... 20
5.4.1 Decimal Literals ... 20
5.4.2 Based Literals ... 21
5.5 Character Literals ... 21
5.6 String Literals ... 21
5.7 Comments ... 22
5.8 Pragmas ... 22
5.9 Reserved Words ... 25

6 Declarations and Types ... 25

6.1 Declarations ... 25
6.2 Types and Subtypes ... 27
6.2.1 Type Declarations ... 28
6.2.2 Subtype Declarations ... 29
6.2.3 Classification of Operations ... 30
6.2.4 Subtype Predicates ... 31
6.3 Objects and Named Numbers ... 34
6.3.1 Object Declarations ... 36
6.3.2 Number Declarations ... 39
6.4 Derived Types and Classes ... 39
6.4.1 Derivation Classes ... 42
6.5 Scalar Types ... 44
6.5.1 Enumeration Types ... 47
6.5.2 Character Types ... 48
6.5.3 Boolean Types ... 49
6.5.4 Integer Types ... 49
6.5.5 Operations of Discrete Types ... 52
6.5.6 Real Types ... 53
6.5.7 Floating Point Types ... 54


AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

6.5.8 Operations of Floating Point Types ... 55
6.5.9 Fixed Point Types ... 56
6.5.10 Operations of Fixed Point Types ... 58
6.6 Array Types ... 59
6.6.1 Index Constraints and Discrete Ranges ... 61
6.6.2 Operations of Array Types ... 62
6.6.3 String Types ... 63
6.7 Discriminants ... 63
6.7.1 Discriminant Constraints ... 66
6.7.2 Operations of Discriminated Types ... 67
6.8 Record Types ... 67
6.8.1 Variant Parts and Discrete Choices ... 70
6.9 Tagged Types and Type Extensions ... 72
6.9.1 Type Extensions ... 75
6.9.2 Dispatching Operations of Tagged Types ... 76
6.9.3 Abstract Types and Subprograms ... 79
6.9.4 Interface Types ... 80
6.10 Access Types ... 82
6.10.1 Incomplete Type Declarations ... 85
6.10.2 Operations of Access Types ... 87
6.11 Declarative Parts ... 93
6.11.1 Completions of Declarations ... 93
7 Names and Expressions ... 94
7.1 Names ... 94
7.1.1 Indexed Components ... 95
7.1.2 Slices ... 96
7.1.3 Selected Components ... 97
7.1.4 Attributes ... 99
7.1.5 User-Defined References ... 100
7.1.6 User-Defined Indexing ... 101
7.2 Literals ... 103
7.2.1 User-Defined Literals ... 104
7.3 Aggregates ... 106
7.3.1 Record Aggregates ... 106
7.3.2 Extension Aggregates ... 108
7.3.3 Array Aggregates ... 110
7.3.4 Delta Aggregates ... 114
7.3.5 Container Aggregates ... 115
7.4 Expressions ... 121
7.5 Operators and Expression Evaluation ... 123
7.5.1 Logical Operators and Short-circuit Control Forms ... 124
7.5.2 Relational Operators and Membership Tests ... 125
7.5.3 Binary Adding Operators ... 128
7.5.4 Unary Adding Operators ... 129
7.5.5 Multiplying Operators ... 129
7.5.6 Highest Precedence Operators ... 132
7.5.7 Conditional Expressions ... 132
7.5.8 Quantified Expressions ... 134
7.5.9 Declare Expressions ... 135
7.5.10 Reduction Expressions ... 136
7.6 Type Conversions ... 139
7.7 Qualified Expressions ... 144
7.8 Allocators ... 144
7.9 Static Expressions and Static Subtypes ... 146
7.9.1 Statically Matching Constraints and Subtypes ... 150

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

7.10 Image Attributes ... 151
8 Statements ... 154
8.1 Simple and Compound Statements - Sequences of Statements ... 155
8.2 Assignment Statements ... 156
8.2.1 Target Name Symbols ... 158
8.3 If Statements ... 158
8.4 Case Statements ... 159
8.5 Loop Statements ... 160
8.5.1 User-Defined Iterator Types ... 164
8.5.2 Generalized Loop Iteration ... 166
8.5.3 Procedural Iterators ... 169
8.6 Block Statements ... 172
8.6.1 Parallel Block Statements ... 172
8.7 Exit Statements ... 174
8.8 Goto Statements ... 175
9 Subprograms ... 175
9.1 Subprogram Declarations ... 176
9.1.1 Preconditions and Postconditions ... 178
9.1.2 The Global and Global'Class Aspects ... 183
9.2 Formal Parameter Modes ... 186
9.3 Subprogram Bodies ... 187
9.3.1 Conformance Rules ... 188
9.3.2 Inline Expansion of Subprograms ... 190
9.4 Subprogram Calls ... 190
9.4.1 Parameter Associations ... 192
9.5 Return Statements ... 194
9.5.1 Nonreturning Subprograms ... 197
9.6 Overloading of Operators ... 198
9.7 Null Procedures ... 199
9.8 Expression Functions ... 199
10 Packages ... 201
10.1 Package Specifications and Declarations ... 201
10.2 Package Bodies ... 202
10.3 Private Types and Private Extensions ... 203
10.3.1 Private Operations ... 205
10.3.2 Type Invariants ... 208
10.3.3 Default Initial Conditions ... 211
10.3.4 Stable Properties of a Type ... 211
10.4 Deferred Constants ... 213
10.5 Limited Types ... 214
10.6 Assignment and Finalization ... 216
10.6.1 Completion and Finalization ... 219
11 Visibility Rules ... 221
11.1 Declarative Region ... 221
11.2 Scope of Declarations ... 222
11.3 Visibility ... 223
11.3.1 Overriding Indicators ... 226
11.4 Use Clauses ... 227
11.5 Renaming Declarations ... 228
11.5.1 Object Renaming Declarations ... 228
11.5.2 Exception Renaming Declarations ... 230
11.5.3 Package Renaming Declarations ... 230
11.5.4 Subprogram Renaming Declarations ... 231

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

11.5.5 Generic Renaming Declarations ....232
11.6 The Context of Overload Resolution ....233
12 Tasks and Synchronization ....235
12.1 Task Units and Task Objects ....236
12.2 Task Execution - Task Activation ....238
12.3 Task Dependence - Termination of Tasks ....239
12.4 Protected Units and Protected Objects ....241
12.5 Intertask Communication ....244
12.5.1 Protected Subprograms and Protected Actions ....248
12.5.2 Entries and Accept Statements ....250
12.5.3 Entry Calls ....253
12.5.4 Requeue Statements ....255
12.6 Delay Statements, Duration, and Time ....257
12.6.1 Formatting, Time Zones, and other operations for Time ....260
12.7 Select Statements ....266
12.7.1 Selective Accept ....266
12.7.2 Timed Entry Calls ....268
12.7.3 Conditional Entry Calls ....269
12.7.4 Asynchronous Transfer of Control ....269
12.8 Abort of a Task - Abort of a Sequence of Statements ....270
12.9 Task and Entry Attributes ....272
12.10 Shared Variables ....272
12.10.1 Conflict Check Policies ....274
12.11 Example of Tasking and Synchronization ....276
13 Program Structure and Compilation Issues ....277
13.1 Separate Compilation ....277
13.1.1 Compilation Units - Library Units ....278
13.1.2 Context Clauses - With Clauses ....281
13.1.3 Subunits of Compilation Units ....283
13.1.4 The Compilation Process ....284
13.1.5 Pragmas and Program Units ....285
13.1.6 Environment-Level Visibility Rules ....286
13.2 Program Execution ....286
13.2.1 Elaboration Control ....288
14 Exceptions ....291
14.1 Exception Declarations ....292
14.2 Exception Handlers ....292
14.3 Raise Statements and Raise Expressions ....293
14.4 Exception Handling ....294
14.4.1 The Package Exceptions ....295
14.4.2 Pragmas Assert and Assertion_Policy ....297
14.4.3 Example of Exception Handling ....299
14.5 Suppressing Checks ....300
14.6 Exceptions and Optimization ....303
15 Generic Units ....304
15.1 Generic Declarations ....304
15.2 Generic Bodies ....306
15.3 Generic Instantiation ....307
15.4 Formal Objects ....309
15.5 Formal Types ....310
15.5.1 Formal Private and Derived Types ....312
15.5.2 Formal Scalar Types ....314
15.5.3 Formal Array Types ....314

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

15.5.4 Formal Access Types ... 315
15.5.5 Formal Interface Types ... 316
15.6 Formal Subprograms ... 316
15.7 Formal Packages ... 319
15.8 Example of a Generic Package ... 321
16 Representation Issues ... 322
16.1 Operational and Representation Aspects ... 322
16.1.1 Aspect Specifications ... 326
16.2 Packed Types ... 329
16.3 Operational and Representation Attributes ... 330
16.4 Enumeration Representation Clauses ... 337
16.5 Record Layout ... 338
16.5.1 Record Representation Clauses ... 338
16.5.2 Storage Place Attributes ... 341
16.5.3 Bit Ordering ... 341
16.6 Change of Representation ... 342
16.7 The Package System ... 342
16.7.1 The Package System.Storage_Elements ... 344
16.7.2 The Package System.Address_To_Access_Conversions ... 345
16.8 Machine Code Insertions ... 346
16.9 Unchecked Type Conversions ... 347
16.9.1 Data Validity ... 348
16.9.2 The Valid Attribute ... 349
16.10 Unchecked Access Value Creation ... 349
16.11 Storage Management ... 350
16.11.1 Storage Allocation Attributes ... 353
16.11.2 Unchecked Storage Deallocation ... 354
16.11.3 Default Storage Pools ... 355
16.11.4 Storage Subpools ... 356
16.11.5 Subpool Reclamation ... 359
16.11.6 Storage Subpool Example ... 359
16.12 Pragma Restrictions and Pragma Profile ... 362
16.12.1 Language-Defined Restrictions and Profiles ... 363
16.13 Streams ... 365
16.13.1 The Streams Subsystem ... 365
16.13.2 Stream-Oriented Attributes ... 368
16.14 Freezing Rules ... 373
Annex A (normative) Predefined Language Environment ... 376
A.1 The Package Standard ... 379
A.2 The Package Ada ... 384
A.3 Character Handling ... 384
A.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters ... 384
A.3.2 The Package Characters.Handling ... 385
A.3.3 The Package Characters.Latin_1 ... 387
A.3.4 The Package Characters.Conversions ... 395
A.3.5 The Package Wide_Characters.Handling ... 397
A.3.6 The Package Wide_Wide_Characters.Handling ... 400
A.4 String Handling ... 400
A.4.1 The Package Strings ... 400
A.4.2 The Package Strings.Maps ... 401
A.4.3 Fixed-Length String Handling ... 404
A.4.4 Bounded-Length String Handling ... 412
A.4.5 Unbounded-Length String Handling ... 419
A.4.6 String-Handling Sets and Mappings ... 425

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

A.4.7 Wide_String Handling ....425
A.4.8 Wide_Wide_String Handling ....428
A.4.9 String Hashing ....430
A.4.10 String Comparison ....431
A.4.11 String Encoding ....432
A.4.12 Universal Text Buffers ....437
A.5 The Numerics Packages ....439
A.5.1 Elementary Functions ....440
A.5.2 Random Number Generation ....443
A.5.3 Attributes of Floating Point Types ....448
A.5.4 Attributes of Fixed Point Types ....452
A.5.5 Big Numbers ....453
A.5.6 Big Integers ....453
A.5.7 Big Reals ....455
A.6 Input-Output ....457
A.7 External Files and File Objects ....457
A.8 Sequential and Direct Files ....458
A.8.1 The Generic Package Sequential_IO ....459
A.8.2 File Management ....460
A.8.3 Sequential Input-Output Operations ....462
A.8.4 The Generic Package Direct_IO ....463
A.8.5 Direct Input-Output Operations ....464
A.9 The Generic Package Storage_IO ....465
A.10 Text Input-Output ....466
A.10.1 The Package Text_IO ....467
A.10.2 Text File Management ....475
A.10.3 Default Input, Output, and Error Files ....475
A.10.4 Specification of Line and Page Lengths ....476
A.10.5 Operations on Columns, Lines, and Pages ....477
A.10.6 Get and Put Procedures ....480
A.10.7 Input-Output of Characters and Strings ....481
A.10.8 Input-Output for Integer Types ....483
A.10.9 Input-Output for Real Types ....485
A.10.10 Input-Output for Enumeration Types ....487
A.10.11 Input-Output for Bounded Strings ....488
A.10.12 Input-Output for Unbounded Strings ....489
A.11 Wide Text Input-Output and Wide Wide Text Input-Output ....491
A.12 Stream Input-Output ....491
A.12.1 The Package Streams.Stream_IO ....491
A.12.2 The Package Text_IO.Text_Streams ....494
A.12.3 The Package Wide_Text_IO.Text_Streams ....495
A.12.4 The Package Wide_Wide_Text_IO.Text_Streams ....495
A.13 Exceptions in Input-Output ....495
A.14 File Sharing ....497
A.15 The Package Command_Line ....497
A.15.1 The Packages Wide_Command_Line and Wide_Wide_Command_Line ....498
A.16 The Package Directories ....498
A.16.1 The Package Directories.Hierarchical_File_Names ....506
A.16.2 The Packages Wide_Directories and Wide_Wide_Directories ....507
A.17 The Package Environment_Variables ....507
A.17.1 The Packages Wide_Environment_Variables and Wide_Wide_Environment_Variables ....509
A.18 Containers ....509
A.18.1 The Package Containers ....511
A.18.2 The Generic Package Containers.Vectors ....511

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

A.18.3 The Generic Package Containers.Doubly_Linked_Lists ... 545
A.18.4 Maps ... 567
A.18.5 The Generic Package Containers.Hashed_Maps ... 577
A.18.6 The Generic Package Containers.Ordered_Maps ... 586
A.18.7 Sets ... 597
A.18.8 The Generic Package Containers.Hashed_Sets ... 608
A.18.9 The Generic Package Containers.Ordered_Sets ... 618
A.18.10 The Generic Package Containers.Multiway_Trees ... 631
A.18.11 The Generic Package Containers.Indefinite_Vectors ... 667
A.18.12 The Generic Package Containers.Indefinite_Doubly_Linked_Lists ... 668
A.18.13 The Generic Package Containers.Indefinite_Hashed_Maps ... 669
A.18.14 The Generic Package Containers.Indefinite_Ordered_Maps ... 669
A.18.15 The Generic Package Containers.Indefinite_Hashed_Sets ... 670
A.18.16 The Generic Package Containers.Indefinite_Ordered_Sets ... 670
A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees ... 670
A.18.18 The Generic Package Containers.Indefinite_Holders ... 671
A.18.19 The Generic Package Containers.Bounded_Vectors ... 676
A.18.20 The Generic Package Containers.Bounded_Doubly_Linked_Lists ... 677
A.18.21 The Generic Package Containers.Bounded_Hashed_Maps ... 679
A.18.22 The Generic Package Containers.Bounded_Ordered_Maps ... 680
A.18.23 The Generic Package Containers.Bounded_Hashed_Sets ... 682
A.18.24 The Generic Package Containers.Bounded_Ordered_Sets ... 684
A.18.25 The Generic Package Containers.Bounded_Multiway_Trees ... 685
A.18.26 Array Sorting ... 687
A.18.27 The Generic Package Containers.Synchronized_Queue_Interfaces ... 689
A.18.28 The Generic Package Containers.Unbounded_Synchronized_Queues ... 690
A.18.29 The Generic Package Containers.Bounded_Synchronized_Queues ... 690
A.18.30 The Generic Package Containers.Unbounded_Priority_Queues ... 691
A.18.31 The Generic Package Containers.Bounded_Priority_Queues ... 693
A.18.32 The Generic Package Containers.Bounded_Indefinite_Holders ... 694
A.18.33 Example of Container Use ... 694
A.19 The Package Locales ... 697

Annex B (normative) Interface to Other Languages ... 698

B.1 Interfacing Aspects ... 698
B.2 The Package Interfaces ... 701
B.3 Interfacing with C and C++ ... 702
B.3.1 The Package Interfaces.C.Strings ... 709
B.3.2 The Generic Package Interfaces.C.Pointers ... 712
B.3.3 Unchecked Union Types ... 714
B.4 Interfacing with COBOL ... 716
B.5 Interfacing with Fortran ... 722

Annex C (informative) Systems Programming ... 725

C.1 Access to Machine Operations ... 725
C.2 Required Representation Support ... 726
C.3 Interrupt Support ... 726
C.3.1 Protected Procedure Handlers ... 728
C.3.2 The Package Interrupts ... 730
C.4 Preelaboration Requirements ... 732
C.5 Aspect Discard_Names ... 732
C.6 Shared Variable Control ... 733
C.6.1 The Package System.Atomic_Operations ... 736
C.6.2 The Package System.Atomic_Operations.Exchange ... 736
C.6.3 The Package System.Atomic_Operations.Test_and_Set ... 737
C.6.4 The Package System.Atomic_Operations.Integer_Arithmetic ... 738

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

C.6.5 The Package System.Atomic_Operations.Modular_Arithmetic ..... 739
C.7 Task Information ..... 740
C.7.1 The Package Task_Identification ..... 740
C.7.2 The Package Task_Attributes ..... 742
C.7.3 The Package Task_Termination ..... 744
Annex D (informative) Real-Time Systems ..... 746
D.1 Task Priorities ..... 746
D.2 Priority Scheduling ..... 748
D.2.1 The Task Dispatching Model ..... 748
D.2.2 Task Dispatching Pragmas ..... 750
D.2.3 Preemptive Dispatching ..... 751
D.2.4 Non-Preemptive Dispatching ..... 752
D.2.5 Round Robin Dispatching ..... 753
D.2.6 Earliest Deadline First Dispatching ..... 754
D.3 Priority Ceiling Locking ..... 757
D.4 Entry Queuing Policies ..... 759
D.4.1 Admission Policies ..... 761
D.5 Dynamic Priorities ..... 761
D.5.1 Dynamic Priorities for Tasks ..... 761
D.5.2 Dynamic Priorities for Protected Objects ..... 762
D.6 Preemptive Abort ..... 763
D.7 Tasking Restrictions ..... 764
D.8 Monotonic Time ..... 767
D.9 Delay Accuracy ..... 770
D.10 Synchronous Task Control ..... 771
D.10.1 Synchronous Barriers ..... 772
D.11 Asynchronous Task Control ..... 773
D.12 Other Optimizations and Determinism Rules ..... 774
D.13 The Ravenscar and Jorvik Profiles ..... 775
D.14 Execution Time ..... 777
D.14.1 Execution Time Timers ..... 779
D.14.2 Group Execution Time Budgets ..... 780
D.14.3 Execution Time of Interrupt Handlers ..... 783
D.15 Timing Events ..... 783
D.16 Multiprocessor Implementation ..... 785
D.16.1 Multiprocessor Dispatching Domains ..... 786
Annex E (informative) Distributed Systems ..... 789
E.1 Partitions ..... 789
E.2 Categorization of Library Units ..... 790
E.2.1 Shared Passive Library Units ..... 791
E.2.2 Remote Types Library Units ..... 792
E.2.3 Remote Call Interface Library Units ..... 793
E.3 Consistency of a Distributed System ..... 794
E.4 Remote Subprogram Calls ..... 795
E.4.1 Asynchronous Remote Calls ..... 796
E.4.2 Example of Use of a Remote Access-to-Class-Wide Type ..... 797
E.5 Partition Communication Subsystem ..... 798
Annex F (informative) Information Systems ..... 801
F.1 Machine_Radix Attribute Definition Clause ..... 801
F.2 The Package Decimal ..... 801
F.3 Edited Output for Decimal Types ..... 802
F.3.1 Picture String Formation ..... 804
F.3.2 Edited Output Generation ..... 807
F.3.3 The Package Text_IO.Editing ..... 811

Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

F.3.4 The Package Wide_Text_IO.Editing ... 814
F.3.5 The Package Wide_Wide_Text_IO.Editing ... 814
Annex G (informative) Numerics ... 815
G.1 Complex Arithmetic ... 815
G.1.1 Complex Types ... 815
G.1.2 Complex Elementary Functions ... 819
G.1.3 Complex Input-Output ... 823
G.1.4 The Package Wide_Text_IO.Complex_IO ... 825
G.1.5 The Package Wide_Wide_Text_IO.Complex_IO ... 825
G.2 Numeric Performance Requirements ... 825
G.2.1 Model of Floating Point Arithmetic ... 826
G.2.2 Model-Oriented Attributes of Floating Point Types ... 827
G.2.3 Model of Fixed Point Arithmetic ... 828
G.2.4 Accuracy Requirements for the Elementary Functions ... 829
G.2.5 Performance Requirements for Random Number Generation ... 831
G.2.6 Accuracy Requirements for Complex Arithmetic ... 831
G.3 Vector and Matrix Manipulation ... 833
G.3.1 Real Vectors and Matrices ... 834
G.3.2 Complex Vectors and Matrices ... 838
Annex H (informative) High Integrity Systems ... 849
H.1 Pragma Normalize_Scalars ... 849
H.2 Documentation of Implementation Decisions ... 850
H.3 Reviewable Object Code ... 850
H.3.1 Pragma Reviewable ... 850
H.3.2 Pragma Inspection_Point ... 851
H.4 High Integrity Restrictions ... 852
H.4.1 Aspect No_Controlled_Parts ... 855
H.5 Pragma Detect_Blocking ... 855
H.6 Pragma Partition_Elaboration_Policy ... 855
H.7 Extensions to Global and Global'Class Aspects ... 856
H.7.1 The Use_Formal and Dispatching Aspects ... 857
Annex I (normative) Obsolescent Features ... 860
I.1 Renamings of Library Units ... 860
I.2 Allowed Replacements of Characters ... 860
I.3 Reduced Accuracy Subtypes ... 861
I.4 The Constrained Attribute ... 861
I.5 ASCII ... 862
I.6 Numeric_Error ... 862
I.7 At Clauses ... 862
I.7.1 Interrupt Entries ... 863
I.8 Mod Clauses ... 864
I.9 The Storage_Size Attribute ... 864
I.10 Specific Suppression of Checks ... 865
I.11 The Class Attribute of Untagged Incomplete Types ... 865
I.12 Pragma Interface ... 865
I.13 Dependence Restriction Identifiers ... 865
I.14 Character and Wide_Character Conversion Functions ... 866
I.15 Aspect-related Pragmas ... 866
I.15.1 Pragma Inline ... 867
I.15.2 Pragma No_Return ... 867
I.15.3 Pragma Pack ... 867
I.15.4 Pragma Storage_Size ... 868
I.15.5 Interfacing Pragmas ... 868
I.15.6 Pragma Unchecked_Union ... 869

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

I.15.7 Pragmas Interrupt_Handler and Attach_Handler ..... 869
I.15.8 Shared Variable Pragmas ..... 870
I.15.9 Pragma CPU ..... 870
I.15.10 Pragma Dispatching_Domain ..... 871
I.15.11 Pragmas Priority and Interrupt_Priority ..... 871
I.15.12 Pragma Relative_Deadline ..... 872
I.15.13 Pragma Asynchronous ..... 872
I.15.14 Elaboration Control Pragmas ..... 872
I.15.15 Distribution Pragmas ..... 873

Annex J (informative) Language-Defined Aspects and Attributes ..... 875

J.1 Language-Defined Aspects ..... 875
J.2 Language-Defined Attributes ..... 879

Annex K (informative) Language-Defined Pragmas ..... 897

Annex L (informative) Summary of Documentation Requirements ..... 899

L.1 Specific Documentation Requirements ..... 899
L.2 Implementation-Defined Characteristics ..... 901
L.3 Implementation Advice ..... 907

Annex M (informative) Syntax Summary ..... 916

M.1 Syntax Rules ..... 916
M.2 Syntax Cross Reference ..... 941

Annex N (informative) Language-Defined Entities ..... 967

N.1 Language-Defined Packages ..... 967
N.2 Language-Defined Types and Subtypes ..... 970
N.3 Language-Defined Subprograms ..... 974
N.4 Language-Defined Exceptions ..... 987
N.5 Language-Defined Objects ..... 988

Bibliography ..... 993

Index ..... 994

Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

# 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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see https://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This fourth edition cancels and replaces the third edition (ISO/IEC 8652:2012), which has been technically revised. It also incorporates the Technical Corrigendum ISO/IEC 8652:2012/Cor.1:2016.

The main changes are as follows:

— improved support for parallel execution is provided via the introduction of parallel loops, parallel blocks, parallel container iteration, and parallel reduction;

— more precise specification of subprogram interfaces is supported via the new aspects Global, Global'Class, and Nonblocking. The Global aspects, in particular, help to determine whether two constructs can safely execute in parallel;

— Pre and Post aspects can now be specified for access-to-subprogram types and for generic formal subprograms; a postcondition for the default initialization of a type can be specified using the new Default_Initial_Condition aspect;

— the behavior of many predefined container operations is now more precisely specified by using pre- and postcondition specifications instead of English descriptions; a restricted ("stable") view for most containers is introduced to support more efficient iteration;

— more flexible uses of static expressions are supported via the introduction of static expression functions along with fewer restrictions on static strings;

— the Image attribute is supported for nonscalar types, and a user-specifiable attribute Put_Image is provided, which determines the value of the Image attribute for a user-defined type;

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

— the use of numeric and string literals is generalized to allow their use with other categories of types, via the new aspects Integer_Literal, Real_Literal, and String_Literal;

— array and record aggregates are made more flexible: index parameters are allowed in an array aggregate to define the components as a function of their array index; discriminants can be defined more flexibly within an aggregate for a variant record type;

— new types of aggregates are provided: delta aggregates to allow the construction of a new object by incremental updates to an existing object; container aggregates to allow construction of an object of a container type by directly specifying its elements;

— a shorthand is provided, using the token '@', to refer to the target of an assignment statement in the expression defining its new value;

— declare expressions are provided that permit the definition and use of local constants or renamings, to allow a large expression to be simplified by defining common parts as named entities;

— support for lightweight iteration is added via the introduction of procedural iterators;

— support for the map-reduce programming strategy is added via the introduction of reduction expressions;

— for constructs that use iterators of any sort, a filter can be specified that restricts the elements produced by the iteration to those that satisfy the condition of the filter;

— predefined packages supporting arbitrary-precision integer and real arithmetic are provided;

— the Jorvik profile is introduced to support hard real-time applications that want to go beyond the restrictions of the Ravenscar profile.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

# Introduction

## Design Goals

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. Subsequent editions, including this fourth edition, have provided further flexibility and added more standardized packages within the framework provided by the 1995 revision.

The need for languages that promote reliability and simplify maintenance is well established. Hence emphasis was placed on program readability over ease of writing. For example, the rules of the language require that program variables be explicitly declared and that their type be specified. Since the type of a variable is invariant, compilers can ensure that operations on variables are compatible with the properties intended for objects of the type. Furthermore, error-prone notations have been avoided, and the syntax of the language avoids the use of encoded forms in favor of more English-like constructs. Finally, the language offers support for separate compilation of program units in a way that facilitates program development and maintenance, and which provides the same degree of checking between units as within a unit.

Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep to a relatively small number of underlying concepts integrated in a consistent and systematic way while continuing to avoid the pitfalls of excessive involution. The design especially aims to provide language constructs that correspond intuitively to the normal expectations of users.

Like many other human activities, the development of programs is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components continues to be a central idea in the design. The concepts of packages, of private types, and of generic units are directly related to this idea, which has ramifications in many other aspects of the language. An allied concern is the maintenance of programs to match changing requirements; type extension and the hierarchical library enable a program to be modified while minimizing disturbance to existing tested and trusted components.

No language can avoid the problem of efficiency. Languages that require over-elaborate compilers, or that lead to the inefficient use of storage or execution time, force these inefficiencies on all machines and on all programs. Every construct of the language was examined in the light of present implementation techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected. Parallel constructs were introduced to simplify making safe and efficient use of modern multicore architectures.

## Language Summary

An Ada program is composed of one or more program units. Program units can be subprograms (which define executable algorithms), packages (which define collections of entities), task units (which define concurrent computations), protected units (which define operations for the coordinated sharing of data between tasks), or generic units (which define parameterized forms of packages and subprograms). Each program unit normally consists of two parts: a specification, containing the information that is visible to other units, and a body, containing the implementation details, which are not visible to other units. Most program units can be compiled separately.

This distinction of the specification and body, and the ability to compile units separately, allows a program to be designed, written, and tested as a set of largely independent software components.

An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. All libraries are structured in a hierarchical manner; this enables the logical decomposition of a subsystem into

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

individual components. The text of a separately compiled program unit names the library units it requires.

*Program Units*

A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the means of invoking a series of actions. For example, it can read data, update variables, or produce some output. It can have parameters, to provide a controlled means of passing information between the procedure and the point of call. A function is the means of invoking the computation of a value. It is similar to a procedure, but in addition will return a result.

A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a set of type declarations and associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package specification.

Subprogram and package units can be compiled separately and arranged in hierarchies of parent and child units giving fine control over visibility of the logical properties and their detailed implementation.

A task unit is the basic unit for defining a task whose sequence of actions can be executed concurrently with those of other tasks. Such tasks can be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit can define either a single executing task or a task type permitting the creation of any number of similar tasks.

A protected unit is the basic unit for defining protected operations for the coordinated use of data shared between tasks. Simple mutual exclusion is provided automatically, and more elaborate sharing protocols can be defined. A protected operation can either be a subprogram or an entry. A protected entry specifies a Boolean expression (an entry barrier) that blocks the execution of the body until it evaluates to True. A protected unit can define a single protected object or a protected type permitting the creation of several similar objects.

*Declarations and Statements*

The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit.

The declarative part associates names with declared entities. For example, a name can denote a type, a constant, a variable, or an exception. A declarative part also introduces the names and parameters of other nested subprograms, packages, task units, protected units, and generic units to be used in the program unit.

The sequence of statements describes a sequence of actions to be performed. The statements are executed in succession (unless a transfer of control causes execution to continue from another place).

An assignment statement changes the value of a variable. A procedure call invokes execution of a procedure after associating any actual parameters provided at the call with the corresponding formal parameters.

Case statements and if statements allow the selection of an enclosed sequence of statements based on the value of an expression or on the value of a condition.

The loop statement provides the basic iterative mechanism in the language. A loop statement specifies a sequence of statements that are executed repeatedly as directed by an iteration scheme, or until an exit statement is encountered.

A block statement comprises a sequence of statements preceded by the declaration of local entities used by the statements.

Certain statements are associated with concurrent execution. A delay statement delays the execution of a task for a specified duration or until a specified time. An entry call statement is written as a procedure call statement; it requests an operation on a task or on a protected object, blocking the caller until the operation can be performed. A called task can accept an entry call by executing a corresponding accept

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

statement, which specifies the actions then to be performed as part of the rendezvous with the calling task. An entry call on a protected object is processed when the corresponding entry barrier evaluates to true, whereupon the body of the entry is executed. The requeue statement permits the provision of a service as a number of related activities with preference control. One form of the select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls and the asynchronous transfer of control in response to some triggering event. Various parallel constructs, including parallel loops and parallel blocks, support the initiation of multiple logical threads of control designed to execute in parallel when multiple processors are available.

Execution of a program unit can encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation can exceed the maximum allowed value of a number, or an attempt can be made to access an array component by using an incorrect index value. To deal with such error situations, the statements of a program unit can be textually followed by exception handlers that specify the actions to be taken when the error situation arises. Exceptions can be raised explicitly by a raise statement.

*Data Types*

Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main categories of types are elementary types (comprising enumeration, numeric, and access types) and composite types (including array and record types).

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.

Numeric types provide a means of performing exact or approximate numerical computations. Exact computations use integer types, which denote sets of consecutive integers. Approximate computations use either fixed point types, with absolute bounds on the error, or floating point types, with relative bounds on the error. The numeric types Integer, Float, and Duration are predefined.

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.

Record, task, and protected types can have special components called discriminants which parameterize the type. Variant record structures that depend on the values of discriminants can be defined within a record type.

Access types allow the construction of linked data structures. A value of an access type represents a reference to an object declared as aliased or to an object created by the evaluation of an allocator. Several variables of an access type can designate the same object, and components of one object can designate the same or other objects. Both the elements in such linked data structures and their relation to other elements can be altered during program execution. Access types also permit references to subprograms to be stored, passed as parameters, and ultimately dereferenced as part of an indirect call.

Private types permit restricted views of a type. A private type can be defined in a package so that only the logically necessary properties are made visible to the users of the type. The full structural details that are externally irrelevant are then only available within the package and any child units.

From any type a new type can be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations can be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives can be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension are 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.

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

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

The concept of a type is further refined by the concept of a subtype, whereby a user can constrain the set of allowed values of a type. Subtypes can be used to define subranges of scalar types, arrays with a limited set of index values, and records and private types with particular discriminant values.

*Other Facilities*

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 must 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.

Aspect clauses can also be used to specify more abstract properties of program entities, such as the pre- and postconditions of a subprogram, or the invariant for a private type. Additional aspects are specifiable to allow user-defined types to use constructs of the language, such as literals, aggregates, or indexing, normally reserved for particular language-defined categories of types, such as numeric types, record types, or array types.

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.

Finally, the language provides a powerful means of parameterization of program units, called generic program units. The generic parameters can be types and subprograms (as well as objects and packages) and so allow general algorithms and data structures to be defined that are applicable to all types of a given class.

Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

# Information technology — Programming Languages — Ada

## 1 Scope

This document specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of computing systems.

This document specifies:

— The form of a program written in Ada;

— The effect of translating and executing such a program;

— The manner in which program units can be combined to form Ada programs;

— The language-defined library units that a conforming implementation is required to supply;

— The permissible variations in conformance to the rules of this document, and the manner in which they are to be documented;

— Those violations of the requirements of this document that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program containing such violations;

— Those violations of the requirements of this document that a conforming implementation is not required to detect.

This document does not specify:

— The means whereby a program written in Ada is transformed into object code executable by a processor;

— The means whereby translation or execution of programs is invoked and the executing units are controlled;

— The size or speed of the object code, or the relative execution speed of different language constructs;

— The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages;

— The effect of unspecified execution;

— The size of a program or program unit that will exceed the capacity of a particular conforming implementation.

## 2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 639-3:2007, *Codes for the representation of names of languages — Part 3: Alpha-3 code for comprehensive coverage of languages*

ISO 3166-1:2020, *Codes for the representation of names of countries and their subdivisions — Part 1: Country codes*

ISO/IEC 10646:2020, *Information technology — Universal coded character set (UCS)*

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

# 3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

## 3.1 Types, objects, and their properties

**3.1.1**
**abstract type**
tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own

**3.1.2**
**access type**
type that has values that designate aliased objects

Note 1 to entry: Access types correspond to "pointer types" or "reference types" in some other languages.

**3.1.3**
**accessibility level**
representation of the lifetime of an entity in terms of the level of dynamic nesting within which the entity is known to exist

**3.1.4**
**aliased view**
view of an object that can be designated by an access value

Note 1 to entry: Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word aliased. The Access attribute can be used to create an access value designating an aliased object.

**3.1.5**
**ancestor of a type**
type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors

Note 1 to entry: Ancestor and descendant are inverse relationships.

**3.1.6**
**array type**
composite type whose components are all of the same type

**3.1.7**
**aspect**
specifiable property of an entity

Note 1 to entry: An aspect can be specified by an `aspect_specification` on the declaration of the entity. Some aspects can be queried via attributes.

**3.1.8**
**attribute**
characteristic or property of an entity that can be queried, and in some cases specified

**3.1.9**
**category of types**
set of types with one or more common properties, such as primitive operations

Note 1 to entry: A category of types that is closed under derivation is also known as a class.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

**3.1.10**
**character type**
enumeration type whose values include characters

**3.1.11**
**class of types**
set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class

Note 1 to entry: The set of types of a class share common properties, such as their primitive operations.

**3.1.12**
**composite type**
type with components, such as an array or record

**3.1.13**
**controlled type**
type that supports user-defined assignment and finalization

Note 1 to entry: Objects are always finalized before being destroyed.

**3.1.14**
**default initial condition**
property that holds for every default-initialized object of a given type

**3.1.15**
**derived type**
type defined in terms of a parent type and zero or more progenitor types given in a derived type definition

Note 1 to entry: A derived type inherits properties such as components and primitive operations from its parent and progenitors.

Note 2 to entry: A type together with the types derived from it (directly or indirectly) form a derivation class.

**3.1.16**
**descendant of a type**
type itself or a type derived (directly or indirectly) from it

Note 1 to entry: Descendant and ancestor are inverse relationships.

**3.1.17**
**discrete type**
type that is either an integer type or an enumeration type

**3.1.18**
**discriminant**
parameter for a composite type, which can control, for example, the bounds of a component that is an array

Note 1 to entry: A discriminant for a task type can be used to pass data to a task of the type upon its creation.

**3.1.19**
**elementary type**
type that does not have components

**3.1.20**
**enumeration type**
type defined by an enumeration of its values, which can be denoted by identifiers or character literals

**3.1.21**
**full type**
type that defines a full view

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

**3.1.22**
**full view**
view of a type that reveals all of its properties

Note 1 to entry: There can be other views of the type that reveal fewer properties.

**3.1.23**
**incomplete type**
type that defines an incomplete view

Note 1 to entry: Incomplete types can be used for defining recursive data structures.

**3.1.24**
**incomplete view**
view of a type that reveals minimal properties

Note 1 to entry: The remaining properties are defined by the full view given elsewhere.

**3.1.25**
**indexable container type**
type that has user-defined behavior for indexing, via the Constant_Indexing or Variable_Indexing aspects

**3.1.26**
**integer type**
type that represents signed or modular integers

Note 1 to entry: A signed integer type has a base range that includes both positive and negative numbers, and has operations that can raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with "wraparound" semantics. Modular types subsume what are called "unsigned types" in some other languages.

**3.1.27**
**interface type**
abstract tagged type that has no components or concrete operations except possibly null procedures

Note 1 to entry: Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type.

**3.1.28**
**invariant**
assertion that is expected to be True for all objects of a given private type when viewed from outside the defining package

**3.1.29**
**iterable container type**
type that has user-defined behavior for iteration, via the Default_Iterator and Iterator_Element aspects

**3.1.30**
**limited type**
type for which copying (such as in an `assignment_statement`) is not allowed

Note 1 to entry: All types are either limited types or nonlimited types.

**3.1.31**
**needed component**
component of a record type or record extension that is required to have its value specified within a given aggregate

**3.1.32**
**nominal subtype**
subtype specified when a view of an object is defined

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

**3.1.33**
**nonlimited type**
type for which copying is allowed

**3.1.34**

**object**
entity that contains a value, and is either a constant or a variable

Note 1 to entry: An object is created by an `object_declaration` or by an `allocator`. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

**3.1.35**
**operational aspect**
aspect that indicates a logical property of an entity, such as the precondition of a subprogram, or the procedure used to write a given type of object to a stream

**3.1.36**
**parent of a derived type**
first ancestor type given in the definition of the derived type

Note 1 to entry: The parent can be almost any kind of type, including an interface type.

**3.1.37**
**partial view**
view of a type that reveals only some of its properties

Note 1 to entry: The remaining properties are defined by the full view given elsewhere.

**3.1.38**
**primitive operations of a type**
operations (such as subprograms) declared together with the type declarations

Note 1 to entry: Primitive operations are inherited by other types in the same derivation class of types.

**3.1.39**
**private extension**
type that extends another type, with the additional properties hidden from its clients

**3.1.40**
**private type**
type that defines a partial view

Note 1 to entry: Private types can be used for defining abstractions that hide unnecessary details from their clients.

**3.1.41**
**progenitor**
type given in the interface list, if any, of an interface, task, protected, or derived type definition

Note 1 to entry: A progenitor is always an interface type.

**3.1.42**
**protected type**
composite type whose components are accessible only through one of its protected operations, which synchronize concurrent access by multiple tasks

**3.1.43**
**real type**
type that has values that are approximations of the real numbers

Note 1 to entry: Floating point and fixed point types are real types.

**3.1.44**
**record extension**
type that extends another type optionally with additional components

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

**3.1.45**
**record type**
composite type consisting of zero or more named components, possibly of different types

**3.1.46**
**reference type**
type that has user-defined behavior for ".all", defined by the Implicit_Dereference aspect

**3.1.47**
**representation aspect**
aspect that indicates how an entity is mapped onto the underlying hardware, for example the size or alignment of an object

**3.1.48**
**scalar type**
either a discrete type or a real type

**3.1.49**
**stable property**
characteristic associated with objects of a given type that is preserved by many of the primitive operations of the type

**3.1.50**
**storage pool object**
object associated with one or more access types from which the storage for objects created by allocators of the access type(s) is obtained

Note 1 to entry: Some storage pools can be partitioned into subpools in order to support finer-grained storage management.

**3.1.51**
**stream**
sequence of elements that can be used, along with the stream-oriented attributes, to support marshalling and unmarshalling of values of most types

**3.1.52**
**subtype**
type together with optional constraints, null exclusions, and predicates, which constrain the values of the type to the subset that satisfies the implied conditions

**3.1.53**
**synchronized**
safely operated on by multiple tasks concurrently

Note 1 to entry: Synchronized is used to qualify entities, as in a synchronized interface.

**3.1.54**
**tagged type**
type whose objects each have a run-time type tag, which indicates the specific type for which the object was originally created

Note 1 to entry: Tagged types can be extended with additional components.

**3.1.55**
**task type**
composite type used to represent active entities which execute concurrently and that can communicate via queued task entries

Note 1 to entry: The top-level task of a partition is called the environment task.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

**3.1.56**
**type**
defining characteristic of each object and expression of the language, with an associated set of values, and a set of primitive operations that implement the fundamental aspects of its semantics

Note 1 to entry: Types are grouped into categories. Most language-defined categories of types are also classes of types.

**3.1.57**
**view of an entity**
representation of an entity that reveals some or all of the properties of the entity

Note 1 to entry: A single entity can have multiple views.

## 3.2 Subprograms and their properties

**3.2.1**
**function**
form of subprogram that returns a result and can be called as part of an expression

**3.2.2**
**overriding operation**
operation that replaces an inherited primitive operation

Note 1 to entry: Operations can be marked explicitly as overriding or not overriding.

**3.2.3**
**postcondition**
assertion that is expected to be True when a given subprogram returns normally

**3.2.4**
**precondition**
assertion that is expected to be True when a given subprogram is called

**3.2.5**
**procedure**
form of subprogram that does not return a result and can only be invoked by a statement

**3.2.6**
**subprogram**
unit of a program that can be brought into execution in various contexts, with the invocation being a subprogram call that can parameterize the effect of the subprogram through the passing of operands

Note 1 to entry: There are two forms of subprograms: functions, which return values, and procedures, which do not.

## 3.3 Other syntactic constructs

**3.3.1**
**aggregate**
construct used to define a value of a composite type by specifying the values of the components of the type

**3.3.2**
**compilation unit**
program unit that is separately compiled

Note 1 to entry: A `compilation_unit` contains either the declaration, the body, or a renaming of a program unit.

**3.3.3**
**construct**
piece of text (explicit or implicit) that is an instance of a syntactic category defined under Syntax

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

**3.3.4**
**container**
structured object that represents a collection of elements all of the same (potentially class-wide) type, such as a vector or a tree

Note 1 to entry: Several predefined container types are provided by the children of package Ada.Containers (see A.18.1).

**3.3.5**
**container aggregate**
construct used to define a value of a type that represents a collection of elements, by explicitly specifying the elements in the collection

**3.3.6**
**core language**
clause or annex in which are defined language constructs or capabilities that are provided by all conforming implementations

Note 1 to entry: A construct is said to be part of the core language if it is defined in a core language clause or annex.

**3.3.7**
**declaration**
language construct that associates a name with (a view of) an entity

Note 1 to entry: A declaration can appear explicitly in the program text (an explicit declaration), or can be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration).

**3.3.8**
**generic instance**
nongeneric unit created by the instantiation of a generic unit

**3.3.9**
**generic unit**
template for a (nongeneric) program unit

Note 1 to entry: The template can be parameterized by objects, types, subprograms, and packages.

Note 2 to entry: Generic units can be used to perform the role that macros sometimes play in other languages.

**3.3.10**
**iterator**
construct that is used to loop over the elements of an array or container

Note 1 to entry: Iterators can be user defined, and can perform arbitrary computations to access elements from a container.

**3.3.11**
**iterator filter**
construct that is used to restrict the elements produced by an iteration to those for which a boolean condition evaluates to True

**3.3.12**
**library unit**
separately compiled program unit, which is a package, a subprogram, or a generic unit

Note 1 to entry: Library units can have other (logically nested) library units as children, and can have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a subsystem.

Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

**3.3.13**
**master construct**
one of certain executable constructs for which there can be objects or tasks whose lifetime ends when the construct completes

Note 1 to entry: Execution of a master construct is a master, with which objects and tasks are associated for the purposes of waiting and finalization.

**3.3.14**
**needed compilation unit**
compilation unit that is necessary to produce an executable partition, because some entity declared or defined within the unit is used elsewhere in the partition

**3.3.15**
**package**
program unit that defines the interface to a group of logically related entities, along with their implementation

Note 1 to entry: Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

**3.3.16**
**parallel construct**
executable construct that defines multiple activities of a single task that can proceed in parallel, via the execution of multiple logical threads of control

**3.3.17**
**partition**
part of a program, which consists of a set of interdependent library units

Note 1 to entry: Each partition can run in a separate address space, possibly on a separate computer. A program can contain just one partition, or it can be distributed across multiple partitions, which can execute concurrently.

**3.3.18**
**pragma**
compiler directive to provide control over and above that provided by the other syntactic constructs of the language

Note 1 to entry: There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation can support additional (implementation-defined) pragmas.

**3.3.19**
**program**
set of partitions, each of which can execute in a separate address space, possibly on a separate computer

**3.3.20**
**program unit**
language construct that is a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal

Note 1 to entry: Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

**3.3.21**
**reduction expression**
expression that defines how to map or transform a collection of values into a new set of values, and then summarize the values by applying an operation to reduce the set to a single value

**3.3.22**
**renaming**
declaration that does not define a new entity, but instead defines a new view of an existing entity

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

**3.3.23**
**specialized needs annex**
annex in which are defined language constructs or capabilities that are not necessarily provided by all conforming implementations

**3.3.24**
**subunit**
body of a program unit that can be compiled separately from its enclosing program unit

## 3.4 Runtime actions

**3.4.1**
**assertion**
boolean expression that is expected to be True at run time at certain specified places

Note 1 to entry: Certain pragmas and aspects define various kinds of assertions.

**3.4.2**
**elaboration**
process by which a declaration achieves its run-time effect

Note 1 to entry: Elaboration is one of the forms of execution.

**3.4.3**
**evaluation**
process by which an expression achieves its run-time effect

Note 1 to entry: Evaluation is one of the forms of execution.

**3.4.4**
**execution**
process by which a construct achieves its run-time effect

Note 1 to entry: Execution of a declaration is also called elaboration. Execution of an expression is also called evaluation.

**3.4.5**
**logical thread of control**
activity within the execution of a program that can proceed in parallel with other activities of the same task, or of separate tasks

**3.4.6**
**master**
execution of a master construct

Note 1 to entry: Each object and task is associated with a master. When a master is left, associated tasks are awaited and associated objects are finalized.

## 3.5 Exceptional situations

**3.5.1**
**check**
test made during execution to determine whether a language rule has been violated

**3.5.2**
**exception**
kind of exceptional situation

**3.5.3**
**exception occurrence**
run-time occurrence of an exceptional situation

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

**3.5.4**
**handle an exception**
perform some actions in response to the arising of an exception

**3.5.5**
**raise an exception**
abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen

**3.5.6**
**suppress a check**
assert that the check cannot fail, and request that the compiler optimize by disabling the check

Note 1 to entry: The compiler is not required to honor this request. Suppressing checks that can fail can cause a program to behave in arbitrary ways.

# 4 General

## 4.1 Structure

This document contains sixteen clauses, fourteen annexes, a bibliography, and an index.

All implementations shall conform to the core of the Ada language, which consists of:

— Clauses 1 to 16

— Annex A, “Predefined Language Environment”

— Annex B, “Interface to Other Languages”

— Annex I, “Obsolescent Features”

In addition, an implementation may conform separately to one or more Specialized Needs Annexes, which define features that are needed by certain application areas:

— Annex C, “Systems Programming”

— Annex D, “Real-Time Systems”

— Annex E, “Distributed Systems”

— Annex F, “Information Systems”

— Annex G, “Numerics”

— Annex H, “High Integrity Systems”

The following Annexes provide additional information:

— Annex J, “Language-Defined Aspects and Attributes”

— Annex K, “Language-Defined Pragmas”

— Annex L, “Summary of Documentation Requirements”

— Annex M, “Syntax Summary”

— Annex N, “Language-Defined Entities”

Each clause is divided into subclauses that have a common structure. Each clause and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

*Syntax*

Syntax rules (indented).

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

*Name Resolution Rules*

Compile-time rules that are used in name resolution, including overload resolution.

*Legality Rules*

Rules that are enforced at compile time. A construct is *legal* if it obeys all of the Legality Rules.

*Static Semantics*

A definition of the compile-time effect of each construct.

*Post-Compilation Rules*

Rules that are enforced before running a partition. A partition is legal if its compilation units are legal and it obeys all of the Post-Compilation Rules.

*Dynamic Semantics*

A definition of the run-time effect of each construct.

*Bounded (Run-Time) Errors*

Situations that result in bounded (run-time) errors (see 4.4).

*Erroneous Execution*

Situations that result in erroneous execution (see 4.4).

*Implementation Requirements*

Additional requirements for conforming implementations.

*Documentation Requirements*

Documentation requirements for conforming implementations.

*Metrics*

Metrics that are specified for the time/space properties of the execution of certain language constructs.

*Implementation Permissions*

Additional permissions given to the implementer.

*Implementation Advice*

Optional advice given to the implementer. It is implementation defined whether or not a given recommendation is obeyed.

NOTE Notes emphasize consequences of the rules described in the (sub)clause or elsewhere.

EXAMPLE Examples illustrate the possible forms of the constructs described.

## 4.2 Conformity of an Implementation

*Implementation Requirements*

A conforming implementation shall:

— Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation;

— Identify all programs or program units that are so large as to exceed the capacity of the implementation (or raise an appropriate exception at run time);

— Identify all programs or program units that contain errors whose detection is required by this document;

— Supply all language-defined library units required by this document;

Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

— Contain no variations except those explicitly permitted by this document, or those that are impossible or impractical to avoid given the implementation's execution environment;

— Specify all such variations in the manner prescribed by this document.

The *external effect* of the execution of an Ada program is defined in terms of its interactions with its external environment. The following are defined as *external interactions*:

— Any interaction with an external file (see A.7);

— The execution of certain `code_statement`s (see 16.8); which `code_statement`s cause external interactions is implementation defined.

— Any call on an imported subprogram (see Annex B), including any parameters passed to it;

— Any result returned or exception propagated from a main subprogram (see 13.2) or an exported subprogram (see Annex B) to an external caller;

— Any read or update of an atomic or volatile object (see C.6);

— The values of imported and exported objects (see Annex B) at the time of any other interaction with the external environment.

A conforming implementation of this document shall produce for the execution of a given Ada program a set of interactions with the external environment whose order and timing are consistent with the definitions and requirements of this document for the semantics of the given program.

An implementation that conforms to this document shall support each capability required by the core language as specified. In addition, an implementation that conforms to this document may conform to one or more Specialized Needs Annexes (or to none). Conformance to a Specialized Needs Annex (Annex C, Annex D, Annex E, Annex F, Annex G, and Annex H) means that each capability required by the Annex shall be provided as specified.

An implementation conforming to this document may provide additional aspects, attributes, library units, and pragmas. However, it shall not provide any aspect, attribute, library unit, or pragma having the same name as an aspect, attribute, library unit, or pragma (respectively) specified in a Specialized Needs Annex unless the provided construct is either as specified in the Specialized Needs Annex or is more limited in capability than that required by the Annex. A program that attempts to use an unsupported capability of an Annex shall either be identified by the implementation before run time or shall raise an exception at run time.

For an implementation that conforms to this document, the implementation of a language-defined unit shall abide by all postconditions, type invariants, and default initial conditions specified for the unit by this document (see 14.4.2).

*Documentation Requirements*

Certain aspects of the semantics are defined to be either *implementation defined* or *unspecified*. In such cases, the set of possible effects is specified, and the implementation may choose any effect in the set. Implementations shall document their behavior in implementation-defined situations, but documentation is not required for unspecified situations. The implementation-defined characteristics are summarized in L.2.

The implementation may choose to document implementation-defined behavior either by documenting what happens in general, or by providing some mechanism for the user to determine what happens in a particular case.

*Implementation Advice*

If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise Program_Error if feasible.

If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit.

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

NOTE The above requirements imply that an implementation conforming to this document can support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities.

## 4.3 Method of Description and Syntax Notation

The form of an Ada program is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules.

The meaning of Ada programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs.

The context-free syntax of the language is described using a simple variant of Backus-Naur Form. In particular:

— Lower case words in a fixed-width font, some containing embedded underlines, are used to denote syntactic categories, for example:

  `case_statement`

— Boldface words are used to denote reserved words, for example:

  **array**

— Square brackets enclose optional items. Thus the two following rules are equivalent.

  `simple_return_statement` ::= **return** [`expression`];
  `simple_return_statement` ::= **return**; | **return** `expression`;

— Curly brackets enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.

  `term` ::= `factor` {`multiplying_operator` `factor`}
  `term` ::= `factor` | `term` `multiplying_operator` `factor`

— A vertical line separates alternative items, for example:

  `constraint` ::= `scalar_constraint` | `composite_constraint`

— For symbols used in this notation (square brackets, curly brackets, and the vertical line), the symbols when surrounded by ' represent themselves, for example:

  `discrete_choice_list` ::= `discrete_choice` {'|' `discrete_choice`}
  `named_container_aggregate` ::=
    '[' `container_element_association_list` ']'

— 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 delimiters, compound delimiters, reserved words, and `numeric_literals` are exclusively made of the characters whose code point is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this document (see 5.1) 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”.

When this document mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using simple upper case mapping, as defined by documents referenced in Clause 2 of ISO/IEC 10646:2020.

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`.

A *construct* is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under “Syntax”.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

A *constituent* of a construct is the construct itself, or any construct appearing within it.

Whenever the run-time semantics defines certain actions to happen in an *arbitrary order*, this means that the implementation shall arrange for these actions to occur in a way that is equivalent to some sequential order, following the rules that result from that sequential order. When evaluations are defined to happen in an arbitrary order, with conversion of the results to some subtypes, or with some runtime checks, the evaluations, conversions, and checks may be arbitrarily interspersed, so long as each expression is evaluated before converting or checking its value. Note that the effect of a program can depend on the order chosen by the implementation. This can happen, for example, if two actual parameters of a given call have side effects.

NOTE 1 The syntax rules describing structured constructs are presented in a form that corresponds to the recommended paragraphing. For example, an `if_statement` is defined as:

```
if_statement ::=
    if condition then
      sequence_of_statements
   {elsif condition then
      sequence_of_statements}
   [else
      sequence_of_statements]
    end if;
```

NOTE 2 The line breaks and indentation in the syntax rules indicate the recommended line breaks and indentation in the corresponding constructs. The preferred places for other line breaks are after semicolons.

## 4.4 Classification of Errors

*Implementation Requirements*

The language definition classifies errors into several different categories:

— Errors that are required to be detected prior to run time by every Ada implementation;

These errors correspond to any violation of a rule given in this document, other than those listed below. In particular, violation of any rule that uses the terms shall, allowed, permitted, legal, or illegal belongs to this category. Any program that contains such an error is not a legal Ada program; on the other hand, the fact that a program is legal does not mean, *per se*, that the program is free from other forms of error.

The rules are further classified as either compile time rules, or post compilation rules, depending on whether a violation has to be detected at the time a compilation unit is submitted to the compiler, or may be postponed until the time a compilation unit is incorporated into a partition of a program.

— Errors that are required to be detected at run time by the execution of an Ada program;

The corresponding error situations are associated with the names of the predefined exceptions. Every Ada compiler is required to generate code that raises the corresponding exception if such an error situation arises during program execution. If such an error situation is certain to arise in every execution of a construct, then an implementation is allowed (although not required) to report this fact at compilation time.

— Bounded errors;

The language rules define certain kinds of errors that are not expected to be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. The errors of this category are called *bounded errors*. The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception Program_Error.

— Erroneous execution.

In addition to bounded errors, the language rules define certain kinds of errors as leading to *erroneous execution*. Like bounded errors, the implementation is not expected to detect such errors

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

either prior to or during run time. Unlike bounded errors, there is no language-specified bound on the possible effect of erroneous execution; the effect is in general not predictable.

*Implementation Permissions*

An implementation may provide *nonstandard modes* of operation. Typically these modes would be selected by a `pragma` or by a command line switch when the compiler is invoked. When operating in a nonstandard mode, the implementation may reject `compilation_units` that do not conform to additional requirements associated with the mode, such as an excessive number of warnings or violation of coding style guidelines. Similarly, in a nonstandard mode, the implementation may apply special optimizations or alternative algorithms that are only meaningful for programs that satisfy certain criteria specified by the implementation. In any case, an implementation shall support a *standard* mode that conforms to the requirements of this document; in particular, in the standard mode, all legal `compilation_units` shall be accepted.

*Implementation Advice*

If an implementation detects a bounded error or erroneous execution, it should raise Program_Error.

# 5 Lexical Elements

The text of a program consists of the texts of one or more `compilations`. The text of a `compilation` is a sequence of lexical elements, each composed of characters; the rules of composition are given in this clause. `Pragmas`, which provide certain information for the compiler, are also described in this clause.

## 5.1 Character Set

The character repertoire for the text of an Ada program consists of the entire coding space described by the ISO/IEC 10646:2020 Universal Coded Character Set. This coding space is organized in *planes*, each plane comprising 65536 characters.

*Syntax*

A `character` is defined by this document for each cell in the coding space described by ISO/IEC 10646:2020, regardless of whether or not ISO/IEC 10646:2020 allocates a character to that cell.

*Static Semantics*

The coded representation for characters is implementation defined (it can be a representation that is not defined within ISO/IEC 10646:2020). A character whose relative code point in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program. The only characters allowed outside of comments are those in categories `other_format`, `format_effector`, and `graphic_character`.

The semantics of an Ada program whose text is not in Normalization Form C (as defined by Clause 22 of ISO/IEC 10646:2020) is implementation defined.

The description of the language definition in this document uses the character properties General Category, Simple Uppercase Mapping, Uppercase Mapping, and Special Case Condition of the documents referenced by Clause 2 of ISO/IEC 10646:2020. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

Characters are categorized as follows:

`letter_uppercase`
: Any character whose General Category is defined to be “Letter, Uppercase”.

`letter_lowercase`
: Any character whose General Category is defined to be “Letter, Lowercase”.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

`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”.

`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”.

`separator_space`
: Any character whose General Category is defined to be “Separator, Space”.

`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”.

`format_effector`
: The characters whose code points 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”.

`graphic_character`
: Any character that is not in the categories `other_control`, `other_private_use`, `other_surrogate`, `format_effector`, and whose relative code point in its plane is neither 16#FFFE# nor 16#FFFF#.

The following names are used when referring to certain characters (the first name is that given in ISO/IEC 10646:2020):

| graphic symbol | name | graphic symbol | name |
|---|---|---|---|
| " | quotation mark | : | colon |
| # | number sign | ; | semicolon |
| & | ampersand | < | less-than sign |
| ' | apostrophe, tick | = | equals sign |

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

| | | | |
|---|---|---|---|
| ( | left parenthesis | > | greater-than sign |
| ) | right parenthesis | _ | low line, underline |
| * | asterisk, multiply | \| | vertical line |
| + | plus sign | / | solidus, divide |
| , | comma | ! | exclamation point |
| – | hyphen-minus, minus | % | percent sign |
| . | full stop, dot, point | [ | left square bracket |
| @ | commercial at, at sign | ] | right square bracket |

*Implementation Requirements*

An Ada implementation shall accept Ada source code in UTF-8 encoding, with or without a BOM (see A.4.11), where every character is represented by its code point. The character pair CARRIAGE RETURN/LINE FEED (code points 16#0D# 16#0A#) signifies a single end of line (see 5.2); every other occurrence of a `format_effector` other than the character whose code point position is 16#09# (CHARACTER TABULATION) also signifies a single end of line.

*Implementation Permissions*

The categories defined above, as well as case mapping and folding, may be based on an implementation-defined version of ISO/IEC 10646 (2003 edition or later).

NOTE The characters in categories `other_control`, `other_private_use`, and `other_surrogate` are only allowed in comments.

## 5.2 Lexical Elements, Separators, and Delimiters

*Static Semantics*

The text of a program consists of the texts of one or more `compilation`s. The text of each `compilation` is a sequence of separate *lexical elements*. Each lexical element is formed from a sequence of characters, and is either a delimiter, an `identifier`, a reserved word, a `numeric_literal`, a `character_literal`, a `string_literal`, or a comment. The meaning of a program depends only on the particular sequences of lexical elements that form its `compilation`s, excluding `comment`s.

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_effector`s other than the character whose code point is 16#09# (CHARACTER TABULATION) signifies at least one end of line.

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:

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

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

— The end of a line is always a separator.

One or more separators are allowed between any two adjacent lexical elements, before the first of each `compilation`, or after the last. At least one separator is required between an `identifier`, a reserved word, or a `numeric_literal` and an adjacent `identifier`, reserved word, or `numeric_literal`.

One or more `other_format` characters are allowed anywhere that a separator is; any such characters have no effect on the meaning of an Ada program.

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

```
& ' ( ) * + , - . / : ; < = > @ [ ] |
```

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

or one of the following *compound delimiters* each composed of two adjacent special characters

=> .. ** := /= >= <= << >> <>

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a `comment`, `string_literal`, `character_literal`, or `numeric_literal`.

The following names are used when referring to compound delimiters:

| delimiter | name |
|---|---|
| => | arrow |
| .. | double dot |
| ** | double star, exponentiate |
| := | assignment (pronounced: "becomes") |
| /= | inequality (pronounced: "not equal") |
| >= | greater than or equal |
| <= | less than or equal |
| << | left label bracket |
| >> | right label bracket |
| <> | box |

*Implementation Requirements*

An implementation shall support lines of at least 200 characters in length, not counting any characters used to signify the end of a line. An implementation shall support lexical elements of at least 200 characters in length. The maximum supported line length and lexical element length are implementation defined.

## 5.3 Identifiers

`Identifiers` are used as names.

*Syntax*

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

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
```

An `identifier` shall not contain two consecutive characters in category `punctuation_connector`, or end with a character in that category.

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

*Legality Rules*

An identifier shall only contain characters that may be present in Normalization Form KC as defined by Clause 22 of ISO/IEC 10646:2020.

*Static Semantics*

Two `identifier`s are considered the same if they consist of the same sequence of characters after applying locale-independent simple case folding, as defined by documents referenced in Clause 2 of ISO/IEC 10646:2020.

After applying simple case folding, an `identifier` shall not be identical to a reserved word.

*Implementation Permissions*

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

NOTE `Identifier`s differing only in the use of corresponding upper and lower case letters are considered the same.

EXAMPLE *Examples of identifiers:*

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

## 5.4 Numeric Literals

There are two kinds of `numeric_literal`s, *real literals* and *integer literals*. A real literal is a `numeric_literal` that includes a point; an integer literal is a `numeric_literal` without a point.

*Syntax*

```
numeric_literal ::= decimal_literal | based_literal
```

NOTE The type of an integer literal is *universal_integer*. The type of a real literal is *universal_real*.

### 5.4.1 Decimal Literals

A `decimal_literal` is a `numeric_literal` in the conventional decimal notation (that is, the base is ten).

*Syntax*

```
decimal_literal ::= numeral [.numeral] [exponent]

numeral ::= digit {[underline] digit}

exponent ::= E [+] numeral | E - numeral

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

An `exponent` for an integer literal shall not have a minus sign.

*Static Semantics*

An underline character in a `numeric_literal` does not affect its meaning. The letter E of an `exponent` can be written either in lower case or in upper case, with the same meaning.

An `exponent` indicates the power of ten by which the value of the `decimal_literal` without the `exponent` is to be multiplied to obtain the value of the `decimal_literal` with the `exponent`.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

EXAMPLE *Examples of decimal literals:*

```
12        0      1E6    123_456     --  integer literals

12.0      0.0    0.456  3.14159_26 --  real literals
```

### 5.4.2 Based Literals

A `based_literal` is a `numeric_literal` expressed in a form that specifies the base explicitly.

*Syntax*

```
based_literal ::=
 base # based_numeral [.based_numeral] # [exponent]

base ::= numeral

based_numeral ::=
 extended_digit {[underline] extended_digit}

extended_digit ::= digit | A | B | C | D | E | F
```

*Legality Rules*

The *base* (the numeric value of the decimal `numeral` preceding the first #) shall be at least two and at most sixteen. The `extended_digits` A through F represent the digits ten through fifteen, respectively. The value of each `extended_digit` of a `based_literal` shall be less than the base.

*Static Semantics*

The conventional meaning of based notation is assumed. An `exponent` indicates the power of the base by which the value of the `based_literal` without the `exponent` is to be multiplied to obtain the value of the `based_literal` with the `exponent`. The `base` and the `exponent`, if any, are in decimal notation.

The `extended_digits` A through F can be written either in lower case or in upper case, with the same meaning.

EXAMPLE *Examples of based literals:*

```
2#1111_1111#  16#FF#       016#0ff#    --  integer literals of value 255
16#E#E1       2#1110_0000#             --  integer literals of value 224
16#F.FF#E+2   2#1.1111_1111_1110#E11   --  real literals of value 4095.0
```

## 5.5 Character Literals

A `character_literal` is formed by enclosing a graphic character between two apostrophe characters.

*Syntax*

```
character_literal ::= 'graphic_character'
```

NOTE A `character_literal` is an enumeration literal of a character type. See 6.5.2.

EXAMPLE *Examples of character literals:*

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

## 5.6 String Literals

A `string_literal` is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets. They are used to represent `operator_symbols` (see 9.1), values of a string type (see 7.2), and array subaggregates (see 7.3.3).

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

*Syntax*

`string_literal ::= "{string_element}"`

`string_element ::= ""` | *non_quotation_mark*`_graphic_character`

A `string_element` is either a pair of quotation marks (""), or a single `graphic_character` other than a quotation mark.

*Static Semantics*

The *sequence of characters* of a `string_literal` is formed from the sequence of `string_elements` between the bracketing quotation marks, in the given order, with a `string_element` that is "" becoming a single quotation mark in the sequence of characters, and any other `string_element` being reproduced in the sequence.

A *null string literal* is a `string_literal` with no `string_elements` between the quotation marks.

NOTE 1 An end of line cannot appear in a `string_literal`.

NOTE 2 No transformation is performed on the sequence of characters of a `string_literal`.

EXAMPLE *Examples of string literals:*

```
"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 ""Εὕρηκα"""
"Volume of cylinder (πr²h) = "
```

## 5.7 Comments

A `comment` starts with two adjacent hyphens and extends up to the end of the line.

*Syntax*

`comment ::= --`{*non_end_of_line*`_character`}

A `comment` may appear on any line of a program.

*Static Semantics*

The presence or absence of `comments` has no influence on whether a program is legal or illegal. Furthermore, `comments` do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

EXAMPLE *Examples of comments:*

```
--  the last sentence above echoes the Algol 68 report

end;  --  processing of Line is complete

--  a long comment can be split onto
--  two or more consecutive lines

----------------  the first two hyphens start the comment
```

## 5.8 Pragmas

A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-defined) pragmas.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

*Syntax*

```
pragma ::=
  pragma identifier
    [(pragma_argument_association {, pragma_argument_association})];

pragma_argument_association ::=
   [pragma_argument_identifier =>] name
  | [pragma_argument_identifier =>] expression
  | pragma_argument_aspect_mark => name
  | pragma_argument_aspect_mark => expression
```

In a `pragma`, any `pragma_argument_associations` without a *pragma_argument*`_identifier` or *pragma_argument*`_aspect_mark` shall precede any associations with a *pragma_argument*`_identifier` or *pragma_argument*`_aspect_mark`.

`Pragma`s are only allowed at the following places in a program:

— After a semicolon delimiter, but not within a `formal_part`, `discriminant_part`, or `declare_expression`.

— At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "`declaration`", "`item`", "`statement`", "`clause`", or "`alternative`", or one of the syntactic categories `variant` or `exception_handler`; but not in place of such a construct if the construct is required, or is part of a list that is required to have at least one such construct.

— In place of a `statement` in a `sequence_of_statements`.

— At any place where a `compilation_unit` is allowed.

Additional syntax rules and placement restrictions exist for specific pragmas.

The *name* of a `pragma` is the identifier following the reserved word **pragma**. The `name` or `expression` of a `pragma_argument_association` is a *pragma argument*.

An *identifier specific to a pragma* is an identifier or reserved word that is used in a pragma argument with special meaning for that pragma.

*Static Semantics*

If an implementation does not recognize the name of a `pragma`, then it has no effect on the semantics of the program. Inside such a `pragma`, the only rules that apply are the Syntax Rules.

*Dynamic Semantics*

Any `pragma` that appears at the place of an executable construct is executed. Unless otherwise specified for a particular pragma, this execution consists of the evaluation of each evaluable pragma argument in an arbitrary order.

*Implementation Requirements*

The implementation shall give a warning message for an unrecognized pragma name.

*Implementation Permissions*

An implementation may provide implementation-defined pragmas; the name of an implementation-defined pragma shall differ from those of the language-defined pragmas.

An implementation may ignore an unrecognized pragma even if it violates some of the Syntax Rules, if detecting the syntax error is too complex.

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

*Implementation Advice*

Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas in a working program are replaced with unrecognized pragmas, the program should still be legal, and should still have the same semantics.

Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows:

— A `pragma` used to complete a declaration;

— A `pragma` used to configure the environment by adding, removing, or replacing `library_item`s.

*Syntax*

The forms of List, Page, and Optimize `pragma`s are as follows:

**pragma** List(`identifier`);

**pragma** Page;

**pragma** Optimize(`identifier`);

Other pragmas are defined throughout this document, and are summarized in Annex K.

*Static Semantics*

A `pragma` List takes one of the `identifier`s On or Off as the single argument. This pragma is allowed anywhere a `pragma` is allowed. It specifies that listing of the compilation is to be continued or suspended until a List `pragma` with the opposite argument is given within the same compilation. The `pragma` itself is always listed if the compiler is producing a listing.

A `pragma` Page is allowed anywhere a `pragma` is allowed. It specifies that the program text which follows the `pragma` should start on a new page (if the compiler is currently producing a listing).

A `pragma` Optimize takes one of the `identifier`s Time, Space, or Off as the single argument. This `pragma` is allowed anywhere a `pragma` is allowed, and it applies until the end of the immediately enclosing declarative region, or for a `pragma` at the place of a `compilation_unit`, to the end of the `compilation`. It gives advice to the implementation as to whether time or space is the primary optimization criterion, or that optional optimizations should be turned off. It is implementation defined how this advice is followed.

EXAMPLE *Examples of pragmas:*

```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Assertion_Policy(Check); -- check assertions
pragma Assert(Exists(File_Name),
              Message => "Nonexistent file"); -- assert file exists
```

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

## 5.9 Reserved Words

*Syntax*

The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case.

**abort**
**abs**
**abstract**
**accept**
**access**
**aliased**
**all**
**and**
**array**
**at**

**begin**
**body**

**case**
**constant**

**declare**
**delay**
**delta**
**digits**
**do**

**else**
**elsif**
**end**
**entry**
**exception**
**exit**

**for**
**function**

**generic**
**goto**

**if**
**in**
**interface**
**is**

**limited**
**loop**

**mod**

**new**
**not**
**null**

**of**
**or**
**others**
**out**
**overriding**

**package**
**parallel**
**pragma**
**private**
**procedure**
**protected**

**raise**
**range**
**record**
**rem**
**renames**
**requeue**

**return**
**reverse**

**select**
**separate**
**some**
**subtype**
**synchronized**

**tagged**
**task**
**terminate**
**then**
**type**

**until**
**use**

**when**
**while**
**with**

**xor**

NOTE The reserved words appear in **lower case boldface** in this document, except when used in the `designator` of an attribute (see 7.1.4). Lower case boldface is also used for a reserved word in a `string_literal` used as an `operator_symbol`. This is merely a convention — programs can be written in whatever typeface is desired and available.

# 6 Declarations and Types

This clause describes the types in the language and the rules for declaring constants, variables, and named numbers.

## 6.1 Declarations

The language defines several kinds of named *entities* that are declared by declarations. The entity's *name* is defined by the declaration, usually by a `defining_identifier`, but sometimes by a `defining_character_literal` or `defining_operator_symbol`. There are also entities that are not directly declared; some of these are elements of other entities, or are allocated dynamically. Such entities can be denoted using `indexed_component`, `selected_component`, or dereference `names` (see 7.1).

There are several forms of declaration. A `basic_declaration` is a form of declaration defined as follows.

*Syntax*

```
basic_declaration ::=
  type_declaration            |subtype_declaration
 |object_declaration          |number_declaration
 |subprogram_declaration      |abstract_subprogram_declaration
 |null_procedure_declaration  |expression_function_declaration
```

Documents
Document Sharing Hub


Full document details: ISO_IEC 8652_2023

```
  |package_declaration        |renaming_declaration
  |exception_declaration      |generic_declaration
  |generic_instantiation
defining_identifier ::= identifier
```

*Static Semantics*

A *declaration* is a language construct that associates a name with (a view of) an entity. A declaration may appear explicitly in the program text (an *explicit* declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an *implicit* declaration).

Each of the following is defined to be a declaration: any `basic_declaration`; an `enumeration_literal_specification`; a `discriminant_specification`; a `component_declaration`; a `defining_identifier` of an `iterated_component_association`; a `loop_parameter_specification`; a `defining_identifier` of a `chunk_specification`; an `iterator_specification`; a `defining_identifier` of an `iterator_parameter_specification`; a `parameter_specification`; a `subprogram_body`; an `extended_return_object_declaration`; an `entry_declaration`; an `entry_index_specification`; a `choice_parameter_specification`; a `generic_formal_parameter_declaration`.

All declarations contain a *definition* for a *view* of an entity. A view consists of an identification of the entity (the entity *of* the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a `renaming_declaration` is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see 11.5)).

When it is clear from context, the term *object* is used in place of *view of an object*. Similarly, the terms *type* and *subtype* are used in place of *view of a type* and *view of a subtype*, respectively.

For each declaration, the language rules define a certain region of text called the *scope* of the declaration (see 11.2). Most declarations associate an `identifier` with a declared entity. Within its scope, and only there, there are places where it is possible to use the `identifier` to refer to the declaration, the view it defines, and the associated entity; these places are defined by the visibility rules (see 11.3). At such places the `identifier` is said to be a *name* of the entity (the `direct_name` or `selector_name`); the name is said to *denote* the declaration, the view, and the associated entity (see 11.6). The declaration is said to *declare* the name, the view, and in most cases, the entity itself.

As an alternative to an `identifier`, an enumeration literal can be declared with a `character_literal` as its name (see 6.5.1), and a function can be declared with an `operator_symbol` as its name (see 9.1).

The syntax rules use the terms `defining_identifier`, `defining_character_literal`, and `defining_operator_symbol` for the defining occurrence of a name; these are collectively called *defining names*. The terms `direct_name` and `selector_name` are used for usage occurrences of `identifier`s, `character_literal`s, and `operator_symbol`s. These are collectively called *usage names*.

*Dynamic Semantics*

The process by which a construct achieves its run-time effect is called *execution*. This process is also called *elaboration* for declarations and *evaluation* for expressions. One of the terms execution, elaboration, or evaluation is defined by this document for each construct that has a run-time effect.

NOTE At compile time, the declaration of an entity *declares* the entity. At run time, the elaboration of the declaration *creates* the entity.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

## 6.2 Types and Subtypes

*Static Semantics*

A *type* is characterized by a set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. An *object* of a given type is a run-time entity that contains (has) a value of the type.

Types are grouped into *categories* of types. There exist several *language-defined categories* of types 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.

The elementary types are the *scalar* types (*discrete* and *real*) and the *access* types (whose values provide access to objects or subprograms). Discrete types are either *integer* types or are defined by enumeration of their values (*enumeration* types). Real types are either *floating point* types or *fixed point* types.

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 6.10.1) 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 10.3) of a type, providing support for data abstraction. The full view (see 6.2.1) of a type represents its complete definition. An incomplete or partial view is considered a composite type, even if the full view is not.

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.

The term *subcomponent* is used in this document 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.

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 6.5 for `range_constraint`s, 6.6.1 for `index_constraint`s, and 6.7.1 for `discriminant_constraint`s. 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 6.10).

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. The other values of the type are *outside* the subtype.

A subtype is called an *unconstrained* subtype if its type has unknown discriminants, or if its type allows range, index, or discriminant constraints, but the subtype does not impose such a constraint; otherwise, the subtype is called a *constrained* subtype (since it has no unconstrained characteristics).

NOTE Any set of types can be called a "category" of types, and any set of types that is closed under derivation (see 6.4) 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 6.2.3), or that correspond to a set of types that are matched by a given kind of generic formal type (see 15.5). 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

AVA Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

classes, the following are examples of "interesting" *language-defined categories*: abstract, incomplete, interface, limited, private, record.

These language-defined categories are organized like this:

- 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

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

### 6.2.1 Type Declarations

A `type_declaration` declares a type and its first subtype.

*Syntax*

```
type_declaration ::= full_type_declaration
 | incomplete_type_declaration
 | private_type_declaration
 | private_extension_declaration

full_type_declaration ::=
  type defining_identifier [known_discriminant_part] is type_definition
    [aspect_specification];
 | task_type_declaration
 | protected_type_declaration

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

Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

```
|derived_type_definition        |interface_type_definition
```

*Legality Rules*

A given type shall not have a subcomponent whose type is the given type itself.

*Static Semantics*

The `defining_identifier` of a `type_declaration` denotes the *first subtype* of the type. The `known_discriminant_part`, if any, defines the discriminants of the type (see 6.7). The remainder of the `type_declaration` defines the remaining characteristics of (the view of) the type.

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 document 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 document sometimes refers to the type of T as simply "the type T".

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.

The definition of a type implicitly declares certain *predefined operators* that operate on the type, according to what classes the type belongs, as specified in 7.5.

The *predefined types* (for example the types Boolean, Wide_Character, Integer, *root_integer*, and *universal_integer*) are the types that are defined in a predefined library package called Standard; this package also includes the (implicit) declarations of their predefined operators. The package Standard is described in A.1.

*Dynamic Semantics*

The elaboration of a `full_type_declaration` consists of the elaboration of the full type definition. Each elaboration of a full type definition creates a distinct type and its first subtype.

EXAMPLE 1 *Examples of type definitions:*

```
(White, Red, Yellow, Green, Blue, Brown, Black)
range 1 .. 72
array(1 .. 10) of Integer
```

EXAMPLE 2 *Examples of type declarations:*

```
type Color  is (White, Red, Yellow, Green, Blue, Brown, Black);
type Column is range 1 .. 72;
type Table  is array(1 .. 10) of Integer;
```

NOTE Each of the above examples declares a named type. The identifier given denotes the first subtype of the type. Other named subtypes of the type can be declared with `subtype_declarations` (see 6.2.2). Although names do not directly denote types, a phrase like "the type Column" is sometimes used in this document to refer to the type of Column, where Column denotes the first subtype of the type. For an example of the definition of an anonymous type, see the declaration of the array Color_Table in 6.3.1; its type is anonymous — it has no nameable subtypes.

### 6.2.2 Subtype Declarations

A `subtype_declaration` declares a subtype of some previously declared type, as defined by a `subtype_indication`.

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

*Syntax*

```
subtype_declaration ::=
 subtype defining_identifier is subtype_indication
   [aspect_specification];

subtype_indication ::= [null_exclusion] subtype_mark [constraint]

subtype_mark ::= subtype_name

constraint ::= scalar_constraint | composite_constraint

scalar_constraint ::=
  range_constraint | digits_constraint | delta_constraint

composite_constraint ::=
  index_constraint | discriminant_constraint
```

*Name Resolution Rules*

A `subtype_mark` shall resolve to denote a subtype. The type *determined by* a `subtype_mark` is the type of the subtype denoted by the `subtype_mark`.

*Dynamic Semantics*

The elaboration of a `subtype_declaration` consists of the elaboration of the `subtype_indication`. The elaboration of a `subtype_indication` creates a new subtype. If the `subtype_indication` does not include a `constraint`, the new subtype has the same (possibly null) constraint as that denoted by the `subtype_mark`. The elaboration of a `subtype_indication` that includes a `constraint` proceeds as follows:

— The `constraint` is first elaborated.

— A check is then made that the `constraint` is *compatible* with the subtype denoted by the `subtype_mark`.

The condition imposed by a `constraint` is the condition obtained after elaboration of the `constraint`. The rules defining compatibility are given for each form of `constraint` in the appropriate subclause. These rules are such that if a `constraint` is *compatible* with a subtype, then the condition imposed by the `constraint` cannot contradict any condition already imposed by the subtype on its values. The exception Constraint_Error is raised if any check of compatibility fails.

NOTE A `scalar_constraint` can be applied to a subtype of an appropriate scalar type (see 6.5, 6.5.9, and I.3), even if the subtype is already constrained. On the other hand, a `composite_constraint` can be applied to a composite subtype (or an access-to-composite subtype) only if the composite subtype is unconstrained (see 6.6.1 and 6.7.1).

EXAMPLE *Examples of subtype declarations:*

```
subtype Rainbow   is Color range Red .. Blue;        --  see 6.2.1
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 6.2.1
subtype Square    is Matrix(1 .. 10, 1 .. 10);       --  see 6.6
subtype Male      is Person(Sex => M);               --  see 6.10.1
subtype Binop_Ref is not null Binop_Ptr;             --  see 6.10
```

### 6.2.3 Classification of Operations

*Static Semantics*

An operation *operates on a type T* if it yields a value of type *T*, if it has an operand whose expected type (see 11.6) is *T*, or if it has an access parameter or access result type (see 9.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test,

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023

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.

The *primitive subprograms* of a specific type are defined as follows:

— The predefined operators of the type (see 7.5);

— For a derived type, the inherited (see 6.4) user-defined subprograms;

— For an enumeration type, the enumeration literals (which are considered parameterless functions — see 6.5.1);

— For a specific type declared immediately within a `package_specification`, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same `package_specification` and that operate on the type;

— For a specific type with an explicitly declared primitive "=" operator whose result type is Boolean, the corresponding "/=" operator (see 9.6);

— 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 11.3) other implicitly declared primitive subprograms of the type.

A primitive subprogram whose designator is an `operator_symbol` is called a *primitive operator*.

### 6.2.4 Subtype Predicates

The language-defined *predicate aspects* Static_Predicate and Dynamic_Predicate may be used to define properties of subtypes. A *predicate specification* is an `aspect_specification` for one of the two predicate aspects. General rules for aspects and `aspect_specification`s are found in Clause 16 (16.1 and 16.1.1 respectively). The predicate aspects are assertion aspects (see 14.4.2). The predicate aspects are not inherited, but their effects are additive, as defined below.

*Name Resolution Rules*

The expected type for a predicate aspect `expression` is any boolean type.

*Static Semantics*

A predicate specification may be given on a `type_declaration` or a `subtype_declaration`, and applies to the declared subtype. In addition, predicate specifications apply to certain other subtypes:

— For a (first) subtype defined by a type declaration, any predicates of parent or progenitor subtypes apply.

— For a subtype created by a `subtype_indication`, the predicate of the subtype denoted by the `subtype_mark` applies.

Predicate checks are defined to be *enabled* or *disabled* for a given subtype as follows:

— If a subtype is declared by a `type_declaration` or `subtype_declaration` that includes a predicate specification, then:

  — if performing checks is required by the Static_Predicate assertion policy (see 14.4.2) and the declaration includes a Static_Predicate specification, then predicate checks are enabled for the subtype;

  — if performing checks is required by the Dynamic_Predicate assertion policy (see 14.4.2) and the declaration includes a Dynamic_Predicate specification, then predicate checks are enabled for the subtype;

  — otherwise, predicate checks are disabled for the subtype, regardless of whether predicate checking is enabled for any other subtypes mentioned in the declaration;

Documents
Document Sharing Hub

© ISO/IEC 2023 – All rights reserved

Full document details: ISO_IEC 8652_2023

— If a subtype is defined by a type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if any predicate checks are enabled for parent or progenitor subtypes;

— If a subtype is created by a `subtype_indication` other than in one of the previous cases, then predicate checks are enabled for the subtype if and only if predicate checks are enabled for the subtype denoted by the `subtype_mark`;

— Otherwise, predicate checks are disabled for the given subtype.

For a subtype with a directly-specified predicate aspect, the following additional language-defined aspect may be specified with an `aspect_specification` (see 16.1.1):

Predicate_Failure
: This aspect shall be specified by an `expression`, which determines the action to be performed when a predicate check fails because a directly-specified predicate aspect of the subtype evaluates to False, as explained below.

*Name Resolution Rules*

The expected type for the Predicate_Failure `expression` is String.

*Legality Rules*

The `expression` of a Static_Predicate specification shall be *predicate-static*; that is, one of the following:

— a static expression;

— a membership test whose *tested*_`simple_expression` is the current instance, and whose `membership_choice_list` meets the requirements for a static membership test (see 7.9);

— a `case_expression` whose *selecting*_`expression` is the current instance, and whose *dependent*_`expression`s are static expressions;

— a call to a predefined equality or ordering operator, where one operand is the current instance, and the other is a static expression;

— a call to a predefined boolean operator **and**, **or**, **xor**, or **not**, where each operand is predicate-static;

— a short-circuit control form where both operands are predicate-static; or

— a parenthesized predicate-static `expression`.

A predicate shall not be specified for an incomplete subtype.

If a predicate applies to a subtype, then that predicate shall not mention any other subtype to which the same predicate applies.

An index subtype, `discrete_range` of an `index_constraint` or `slice`, or a `discrete_subtype_definition` of a `constrained_array_definition`, `entry_declaration`, or `entry_index_specification` shall not denote a subtype to which predicate specifications apply.

The `prefix` of an `attribute_reference` whose `attribute_designator` is First, Last, or Range shall not denote a scalar subtype to which predicate specifications apply.

The `discrete_subtype_definition` of a `loop_parameter_specification` shall not denote a nonstatic subtype to which predicate specifications apply or any subtype to which Dynamic_Predicate specifications apply.

The `discrete_choice` of a `named_array_aggregate` shall not denote a nonstatic subtype to which predicate specifications apply.

In addition to the places where Legality Rules normally apply (see 15.3), these rules apply also in the private part of an instance of a generic unit.

AVA Documents
Document Sharing Hub


© ISO/IEC 2023 – All rights reserved
Full document details: ISO_IEC 8652_2023