BS ISO/IEC 8652:2012

BSI Standards Publication

Information technology — Programming languages — Ada

bsi.

...making excellence a habit.™

Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO/IEC 8652:2012. It supersedes BS ISO/IEC 8652:1995 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/5/-/9, Ada Panel.

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

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2013. Published by BSI Standards Limited 2013

ISBN 978 0 580 79670 8

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 May 2013.

Amendments issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 8652:2012

BS ISO/IEC 8652:2012

INTERNATIONAL STANDARD

ISO/IEC 8652

Third edition
2012-12-15

Information technology — Programming languages — Ada

Technologies de l'information — Langages de programmation — Ada

Reference number
ISO/IEC 8652:2012(E)

Documents
Document Sharing Hub

© ISO/IEC 2012

Full document details: BS ISO/IEC 8652:2012

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2012

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

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Documents
Document Sharing Hub
© ISO/IEC 2012 – All rights reserved

Full document details: BS ISO/IEC 8652:2012

Table of Contents

Table of Contents i
Foreword xi
Introduction xii
1 General 1
1.1 Scope 1
1.1.1 Extent 1
1.1.2 Structure 2
1.1.3 Conformity of an Implementation with the Standard 4
1.1.4 Method of Description and Syntax Notation 5
1.1.5 Classification of Errors 6
1.2 Normative References 7
1.3 Terms and Definitions 8
2 Lexical Elements 9
2.1 Character Set 9
2.2 Lexical Elements, Separators, and Delimiters 11
2.3 Identifiers 12
2.4 Numeric Literals 13
2.4.1 Decimal Literals 13
2.4.2 Based Literals 13
2.5 Character Literals 14
2.6 String Literals 14
2.7 Comments 15
2.8 Pragmas 15
2.9 Reserved Words 17
3 Declarations and Types 19
3.1 Declarations 19
3.2 Types and Subtypes 20
3.2.1 Type Declarations 21
3.2.2 Subtype Declarations 23
3.2.3 Classification of Operations 24
3.2.4 Subtype Predicates 24
3.3 Objects and Named Numbers 26
3.3.1 Object Declarations 28
3.3.2 Number Declarations 31
3.4 Derived Types and Classes 31
3.4.1 Derivation Classes 34
3.5 Scalar Types 35
3.5.1 Enumeration Types 40
3.5.2 Character Types 41
3.5.3 Boolean Types 42
3.5.4 Integer Types 42
3.5.5 Operations of Discrete Types 44
3.5.6 Real Types 45
3.5.7 Floating Point Types 46
3.5.8 Operations of Floating Point Types 48
3.5.9 Fixed Point Types 48
3.5.10 Operations of Fixed Point Types 50

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

3.6 Array Types 51
3.6.1 Index Constraints and Discrete Ranges 54
3.6.2 Operations of Array Types 55
3.6.3 String Types 56
3.7 Discriminants 56
3.7.1 Discriminant Constraints 59
3.7.2 Operations of Discriminated Types 60
3.8 Record Types 60
3.8.1 Variant Parts and Discrete Choices 62
3.9 Tagged Types and Type Extensions 64
3.9.1 Type Extensions 67
3.9.2 Dispatching Operations of Tagged Types 68
3.9.3 Abstract Types and Subprograms 71
3.9.4 Interface Types 72
3.10 Access Types 75
3.10.1 Incomplete Type Declarations 77
3.10.2 Operations of Access Types 79
3.11 Declarative Parts 85
3.11.1 Completions of Declarations 85
4 Names and Expressions 87
4.1 Names 87
4.1.1 Indexed Components 88
4.1.2 Slices 89
4.1.3 Selected Components 89
4.1.4 Attributes 91
4.1.5 User-Defined References 92
4.1.6 User-Defined Indexing 93
4.2 Literals 95
4.3 Aggregates 96
4.3.1 Record Aggregates 96
4.3.2 Extension Aggregates 98
4.3.3 Array Aggregates 99
4.4 Expressions 102
4.5 Operators and Expression Evaluation 103
4.5.1 Logical Operators and Short-circuit Control Forms 104
4.5.2 Relational Operators and Membership Tests 105
4.5.3 Binary Adding Operators 109
4.5.4 Unary Adding Operators 110
4.5.5 Multiplying Operators 110
4.5.6 Highest Precedence Operators 112
4.5.7 Conditional Expressions 113
4.5.8 Quantified Expressions 114
4.6 Type Conversions 115
4.7 Qualified Expressions 119
4.8 Allocators 120
4.9 Static Expressions and Static Subtypes 122
4.9.1 Statically Matching Constraints and Subtypes 125
5 Statements 127
5.1 Simple and Compound Statements - Sequences of Statements 127
5.2 Assignment Statements 128
5.3 If Statements 129
5.4 Case Statements 130

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

5.5 Loop Statements ... 131
5.5.1 User-Defined Iterator Types ... 133
5.5.2 Generalized Loop Iteration ... 134
5.6 Block Statements ... 136
5.7 Exit Statements ... 137
5.8 Goto Statements ... 137
6 Subprograms ... 139
6.1 Subprogram Declarations ... 139
6.1.1 Preconditions and Postconditions ... 142
6.2 Formal Parameter Modes ... 144
6.3 Subprogram Bodies ... 145
6.3.1 Conformance Rules ... 146
6.3.2 Inline Expansion of Subprograms ... 148
6.4 Subprogram Calls ... 148
6.4.1 Parameter Associations ... 150
6.5 Return Statements ... 153
6.5.1 Nonreturning Procedures ... 155
6.6 Overloading of Operators ... 156
6.7 Null Procedures ... 157
6.8 Expression Functions ... 158
7 Packages ... 159
7.1 Package Specifications and Declarations ... 159
7.2 Package Bodies ... 160
7.3 Private Types and Private Extensions ... 161
7.3.1 Private Operations ... 163
7.3.2 Type Invariants ... 165
7.4 Deferred Constants ... 167
7.5 Limited Types ... 168
7.6 Assignment and Finalization ... 170
7.6.1 Completion and Finalization ... 172
8 Visibility Rules ... 175
8.1 Declarative Region ... 175
8.2 Scope of Declarations ... 176
8.3 Visibility ... 177
8.3.1 Overriding Indicators ... 179
8.4 Use Clauses ... 180
8.5 Renaming Declarations ... 181
8.5.1 Object Renaming Declarations ... 182
8.5.2 Exception Renaming Declarations ... 183
8.5.3 Package Renaming Declarations ... 183
8.5.4 Subprogram Renaming Declarations ... 184
8.5.5 Generic Renaming Declarations ... 186
8.6 The Context of Overload Resolution ... 186
9 Tasks and Synchronization ... 189
9.1 Task Units and Task Objects ... 189
9.2 Task Execution - Task Activation ... 192
9.3 Task Dependence - Termination of Tasks ... 193
9.4 Protected Units and Protected Objects ... 194
9.5 Intertask Communication ... 197
9.5.1 Protected Subprograms and Protected Actions ... 199
9.5.2 Entries and Accept Statements ... 200

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

9.5.3 Entry Calls 203
9.5.4 Requeue Statements 205
9.6 Delay Statements, Duration, and Time 207
9.6.1 Formatting, Time Zones, and other operations for Time 209
9.7 Select Statements 215
9.7.1 Selective Accept 215
9.7.2 Timed Entry Calls 217
9.7.3 Conditional Entry Calls 218
9.7.4 Asynchronous Transfer of Control 219
9.8 Abort of a Task - Abort of a Sequence of Statements 220
9.9 Task and Entry Attributes 221
9.10 Shared Variables 222
9.11 Example of Tasking and Synchronization 223
10 Program Structure and Compilation Issues 225
10.1 Separate Compilation 225
10.1.1 Compilation Units - Library Units 225
10.1.2 Context Clauses - With Clauses 228
10.1.3 Subunits of Compilation Units 230
10.1.4 The Compilation Process 232
10.1.5 Pragmas and Program Units 233
10.1.6 Environment-Level Visibility Rules 234
10.2 Program Execution 234
10.2.1 Elaboration Control 236
11 Exceptions 241
11.1 Exception Declarations 241
11.2 Exception Handlers 242
11.3 Raise Statements 243
11.4 Exception Handling 243
11.4.1 The Package Exceptions 244
11.4.2 Pragmas Assert and Assertion_Policy 246
11.4.3 Example of Exception Handling 248
11.5 Suppressing Checks 249
11.6 Exceptions and Optimization 252
12 Generic Units 253
12.1 Generic Declarations 253
12.2 Generic Bodies 255
12.3 Generic Instantiation 256
12.4 Formal Objects 258
12.5 Formal Types 259
12.5.1 Formal Private and Derived Types 261
12.5.2 Formal Scalar Types 263
12.5.3 Formal Array Types 263
12.5.4 Formal Access Types 264
12.5.5 Formal Interface Types 265
12.6 Formal Subprograms 265
12.7 Formal Packages 267
12.8 Example of a Generic Package 269
13 Representation Issues 273
13.1 Operational and Representation Aspects 273
13.1.1 Aspect Specifications 276
13.2 Packed Types 278

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

13.3 Operational and Representation Attributes 279
13.4 Enumeration Representation Clauses 285
13.5 Record Layout 286
13.5.1 Record Representation Clauses 286
13.5.2 Storage Place Attributes 288
13.5.3 Bit Ordering 289
13.6 Change of Representation 290
13.7 The Package System 291
13.7.1 The Package System.Storage_Elements 293
13.7.2 The Package System.Address_To_Access_Conversions 294
13.8 Machine Code Insertions 294
13.9 Unchecked Type Conversions 295
13.9.1 Data Validity 296
13.9.2 The Valid Attribute 297
13.10 Unchecked Access Value Creation 298
13.11 Storage Management 298
13.11.1 Storage Allocation Attributes 301
13.11.2 Unchecked Storage Deallocation 302
13.11.3 Default Storage Pools 303
13.11.4 Storage Subpools 304
13.11.5 Subpool Reclamation 306
13.11.6 Storage Subpool Example 307
13.12 Pragma Restrictions and Pragma Profile 309
13.12.1 Language-Defined Restrictions and Profiles 310
13.13 Streams 312
13.13.1 The Package Streams 312
13.13.2 Stream-Oriented Attributes 313
13.14 Freezing Rules 318

The Standard Libraries 321

Annex A (normative) **Predefined Language Environment** 323

A.1 The Package Standard 326
A.2 The Package Ada 330
A.3 Character Handling 330
A.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters 330
A.3.2 The Package Characters.Handling 331
A.3.3 The Package Characters.Latin_1 333
A.3.4 The Package Characters.Conversions 338
A.3.5 The Package Wide_Characters.Handling 340
A.3.6 The Package Wide_Wide_Characters.Handling 342
A.4 String Handling 343
A.4.1 The Package Strings 343
A.4.2 The Package Strings.Maps 343
A.4.3 Fixed-Length String Handling 346
A.4.4 Bounded-Length String Handling 354
A.4.5 Unbounded-Length String Handling 361
A.4.6 String-Handling Sets and Mappings 366
A.4.7 Wide_String Handling 366
A.4.8 Wide_Wide_String Handling 368
A.4.9 String Hashing 371
A.4.10 String Comparison 372
A.4.11 String Encoding 373
A.5 The Numerics Packages 378

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

A.5.1 Elementary Functions 378
A.5.2 Random Number Generation 381
A.5.3 Attributes of Floating Point Types 386
A.5.4 Attributes of Fixed Point Types 390
A.6 Input-Output 390
A.7 External Files and File Objects 390
A.8 Sequential and Direct Files 391
A.8.1 The Generic Package Sequential_IO 392
A.8.2 File Management 393
A.8.3 Sequential Input-Output Operations 395
A.8.4 The Generic Package Direct_IO 395
A.8.5 Direct Input-Output Operations 396
A.9 The Generic Package Storage_IO 397
A.10 Text Input-Output 397
A.10.1 The Package Text_IO 399
A.10.2 Text File Management 403
A.10.3 Default Input, Output, and Error Files 404
A.10.4 Specification of Line and Page Lengths 405
A.10.5 Operations on Columns, Lines, and Pages 406
A.10.6 Get and Put Procedures 409
A.10.7 Input-Output of Characters and Strings 410
A.10.8 Input-Output for Integer Types 412
A.10.9 Input-Output for Real Types 414
A.10.10 Input-Output for Enumeration Types 416
A.10.11 Input-Output for Bounded Strings 417
A.10.12 Input-Output for Unbounded Strings 418
A.11 Wide Text Input-Output and Wide Wide Text Input-Output 419
A.12 Stream Input-Output 420
A.12.1 The Package Streams.Stream_IO 420
A.12.2 The Package Text_IO.Text_Streams 422
A.12.3 The Package Wide_Text_IO.Text_Streams 423
A.12.4 The Package Wide_Wide_Text_IO.Text_Streams 423
A.13 Exceptions in Input-Output 423
A.14 File Sharing 425
A.15 The Package Command_Line 425
A.16 The Package Directories 426
A.16.1 The Package Directories.Hierarchical_File_Names 433
A.17 The Package Environment_Variables 435
A.18 Containers 438
A.18.1 The Package Containers 438
A.18.2 The Generic Package Containers.Vectors 438
A.18.3 The Generic Package Containers.Doubly_Linked_Lists 454
A.18.4 Maps 465
A.18.5 The Generic Package Containers.Hashed_Maps 471
A.18.6 The Generic Package Containers.Ordered_Maps 475
A.18.7 Sets 479
A.18.8 The Generic Package Containers.Hashed_Sets 486
A.18.9 The Generic Package Containers.Ordered_Sets 491
A.18.10 The Generic Package Containers.Multiway_Trees 496
A.18.11 The Generic Package Containers.Indefinite_Vectors 510
A.18.12 The Generic Package Containers.Indefinite_Doubly_Linked_Lists 510
A.18.13 The Generic Package Containers.Indefinite_Hashed_Maps 511
A.18.14 The Generic Package Containers.Indefinite_Ordered_Maps 511
A.18.15 The Generic Package Containers.Indefinite_Hashed_Sets 511

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

A.18.16 The Generic Package Containers.Indefinite_Ordered_Sets 512
A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees 512
A.18.18 The Generic Package Containers.Indefinite_Holders..... 512
A.18.19 The Generic Package Containers.Bounded_Vectors 516
A.18.20 The Generic Package Containers.Bounded_Doubly_Linked_Lists 516
A.18.21 The Generic Package Containers.Bounded_Hashed_Maps 518
A.18.22 The Generic Package Containers.Bounded_Ordered_Maps 519
A.18.23 The Generic Package Containers.Bounded_Hashed_Sets..... 520
A.18.24 The Generic Package Containers.Bounded_Ordered_Sets..... 521
A.18.25 The Generic Package Containers.Bounded_Multiway_Trees..... 522
A.18.26 Array Sorting 524
A.18.27 The Generic Package Containers.Synchronized_Queue_Interfaces 525
A.18.28 The Generic Package Containers.Unbounded_Synchronized_Queues .. 526
A.18.29 The Generic Package Containers.Bounded_Synchronized_Queues..... 527
A.18.30 The Generic Package Containers.Unbounded_Priority_Queues 527
A.18.31 The Generic Package Containers.Bounded_Priority_Queues..... 529
A.18.32 Example of Container Use 530
A.19 The Package Locales 532

Annex B (normative) **Interface to Other Languages**.....533

B.1 Interfacing Aspects 533
B.2 The Package Interfaces 536
B.3 Interfacing with C and C++ 537
B.3.1 The Package Interfaces.C.Strings 543
B.3.2 The Generic Package Interfaces.C.Pointers..... 546
B.3.3 Unchecked Union Types 548
B.4 Interfacing with COBOL..... 550
B.5 Interfacing with Fortran 556

Annex C (normative) **Systems Programming** 559

C.1 Access to Machine Operations 559
C.2 Required Representation Support..... 560
C.3 Interrupt Support..... 560
C.3.1 Protected Procedure Handlers 562
C.3.2 The Package Interrupts 564
C.4 Preelaboration Requirements 566
C.5 Pragma Discard_Names 566
C.6 Shared Variable Control 567
C.7 Task Information 569
C.7.1 The Package Task_Identification 569
C.7.2 The Package Task_Attributes..... 571
C.7.3 The Package Task_Termination 573

Annex D (normative) **Real-Time Systems** 575

D.1 Task Priorities 575
D.2 Priority Scheduling 577
D.2.1 The Task Dispatching Model 577
D.2.2 Task Dispatching Pragmas 578
D.2.3 Preemptive Dispatching 580
D.2.4 Non-Preemptive Dispatching..... 580
D.2.5 Round Robin Dispatching..... 582
D.2.6 Earliest Deadline First Dispatching..... 583
D.3 Priority Ceiling Locking 585
D.4 Entry Queuing Policies 587
D.5 Dynamic Priorities..... 588

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

D.5.1 Dynamic Priorities for Tasks 588
D.5.2 Dynamic Priorities for Protected Objects 589
D.6 Preemptive Abort 590
D.7 Tasking Restrictions 591
D.8 Monotonic Time 593
D.9 Delay Accuracy 596
D.10 Synchronous Task Control 597
D.10.1 Synchronous Barriers 598
D.11 Asynchronous Task Control 599
D.12 Other Optimizations and Determinism Rules 600
D.13 The Ravenscar Profile 601
D.14 Execution Time 602
D.14.1 Execution Time Timers 604
D.14.2 Group Execution Time Budgets 606
D.14.3 Execution Time of Interrupt Handlers 608
D.15 Timing Events 608
D.16 Multiprocessor Implementation 610
D.16.1 Multiprocessor Dispatching Domains 611

Annex E (normative) **Distributed Systems** 615

E.1 Partitions 615
E.2 Categorization of Library Units 617
E.2.1 Shared Passive Library Units 617
E.2.2 Remote Types Library Units 618
E.2.3 Remote Call Interface Library Units 619
E.3 Consistency of a Distributed System 620
E.4 Remote Subprogram Calls 621
E.4.1 Asynchronous Remote Calls 623
E.4.2 Example of Use of a Remote Access-to-Class-Wide Type 623
E.5 Partition Communication Subsystem 625

Annex F (normative) **Information Systems** 629

F.1 Machine_Radix Attribute Definition Clause 629
F.2 The Package Decimal 629
F.3 Edited Output for Decimal Types 630
F.3.1 Picture String Formation 632
F.3.2 Edited Output Generation 635
F.3.3 The Package Text_IO.Editing 638
F.3.4 The Package Wide_Text_IO.Editing 641
F.3.5 The Package Wide_Wide_Text_IO.Editing 642

Annex G (normative) **Numerics** 643

G.1 Complex Arithmetic 643
G.1.1 Complex Types 643
G.1.2 Complex Elementary Functions 647
G.1.3 Complex Input-Output 651
G.1.4 The Package Wide_Text_IO.Complex_IO 653
G.1.5 The Package Wide_Wide_Text_IO.Complex_IO 653
G.2 Numeric Performance Requirements 653
G.2.1 Model of Floating Point Arithmetic 654
G.2.2 Model-Oriented Attributes of Floating Point Types 655
G.2.3 Model of Fixed Point Arithmetic 656
G.2.4 Accuracy Requirements for the Elementary Functions 658
G.2.5 Performance Requirements for Random Number Generation 659
G.2.6 Accuracy Requirements for Complex Arithmetic 660

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

G.3 Vector and Matrix Manipulation ... 662
G.3.1 Real Vectors and Matrices ... 662
G.3.2 Complex Vectors and Matrices ... 667
Annex H (normative) High Integrity Systems ... 677
H.1 Pragma Normalize_Scalars ... 677
H.2 Documentation of Implementation Decisions ... 678
H.3 Reviewable Object Code ... 678
H.3.1 Pragma Reviewable ... 678
H.3.2 Pragma Inspection_Point ... 679
H.4 High Integrity Restrictions ... 680
H.5 Pragma Detect_Blocking ... 682
H.6 Pragma Partition_Elaboration_Policy ... 682
Annex J (normative) Obsolescent Features ... 685
J.1 Renamings of Library Units ... 685
J.2 Allowed Replacements of Characters ... 685
J.3 Reduced Accuracy Subtypes ... 686
J.4 The Constrained Attribute ... 686
J.5 ASCII ... 687
J.6 Numeric_Error ... 687
J.7 At Clauses ... 687
J.7.1 Interrupt Entries ... 688
J.8 Mod Clauses ... 689
J.9 The Storage_Size Attribute ... 689
J.10 Specific Suppression of Checks ... 689
J.11 The Class Attribute of Untagged Incomplete Types ... 690
J.12 Pragma Interface ... 690
J.13 Dependence Restriction Identifiers ... 690
J.14 Character and Wide_Character Conversion Functions ... 691
J.15 Aspect-related Pragmas ... 691
J.15.1 Pragma Inline ... 691
J.15.2 Pragma No_Return ... 692
J.15.3 Pragma Pack ... 692
J.15.4 Pragma Storage_Size ... 692
J.15.5 Interfacing Pragmas ... 692
J.15.6 Pragma Unchecked_Union ... 693
J.15.7 Pragmas Interrupt_Handler and Attach_Handler ... 694
J.15.8 Shared Variable Pragmas ... 694
J.15.9 Pragma CPU ... 695
J.15.10 Pragma Dispatching_Domain ... 695
J.15.11 Pragmas Priority and Interrupt_Priority ... 696
J.15.12 Pragma Relative_Deadline ... 696
J.15.13 Pragma Asynchronous ... 697
Annex K (informative) Language-Defined Aspects and Attributes ... 699
K.1 Language-Defined Aspects ... 699
K.2 Language-Defined Attributes ... 702
Annex L (informative) Language-Defined Pragmas ... 717
Annex M (informative) Summary of Documentation Requirements ... 719
M.1 Specific Documentation Requirements ... 719
M.2 Implementation-Defined Characteristics ... 721
M.3 Implementation Advice ... 726

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

Annex N (informative) **Glossary** **735**

Annex P (informative) **Syntax Summary** **741**

Annex Q (informative) **Language-Defined Entities** **769**

Q.1 Language-Defined Packages **769**
Q.2 Language-Defined Types and Subtypes **771**
Q.3 Language-Defined Subprograms **776**
Q.4 Language-Defined Exceptions **785**
Q.5 Language-Defined Objects **786**

Index **791**

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

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.

International Standard ISO/IEC 8652 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology* Subcommittee SC22, *Programming languages, their environments and system software interfaces*.

This third edition cancels and replaces the second edition (ISO/IEC 8652:1995), which has been technically revised. It also incorporates the Technical Corrigendum ISO/IEC 8652:1995:COR.1:2001 and Amendment ISO/IEC 8652:1995:AMD 1:2007.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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. This third edition provides further flexibility and adds 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.

Language Summary

An Ada program is composed of one or more program units. Program units may 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 must be visible to other units, and a body, containing the implementation details, which need not be 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

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

individual components. The text of a separately compiled program unit must name 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 may read data, update variables, or produce some output. It may 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 may 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 may be executed concurrently with those of other tasks. Such tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit may 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 must be True before the body of the entry is executed. A protected unit may 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 may 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 that are 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 that a sequence of statements is to be 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.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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 may accept an entry call by executing a corresponding accept 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.

Execution of a program unit may encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation may exceed the maximum allowed value of a number, or an attempt may 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 classes 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 may 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 may designate the same object, and components of one object may 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 may be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations may be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

may be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension must be tagged, so that the specific type of an operand within a derivation class can be identified at run time. When an operation of a tagged type is applied to an operand whose specific type is not known until run time, implicit dispatching is performed based on the tag of the operand.

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

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 are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code.

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.

Language Changes

This International Standard replaces the second edition of 1995. It modifies the previous edition by making changes and additions that improve the capability of the language and the reliability of programs written in the language. This edition incorporates the changes from Amendment 1 (ISO/IEC 8652:1995:AMD 1:2007), which were designed to improve the portability of programs, interfacing to other languages, and both the object-oriented and real-time capabilities.

Significant changes originating in Amendment 1 are incorporated:

- Support for program text is extended to cover the entire ISO/IEC 10646:2003 repertoire. Execution support now includes the 32-bit character set. See subclauses 2.1, 3.5.2, 3.6.3, A.1, A.3, and A.4.
- The object-oriented model has been improved by the addition of an interface facility which provides multiple inheritance and additional flexibility for type extensions. See subclauses 3.4, 3.9, and 7.3. An alternative notation for calling operations more akin to that used in other languages has also been added. See subclause 4.1.3.
- Access types have been further extended to unify properties such as the ability to access constants and to exclude null values. See clause 3.10. Anonymous access types are now

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

permitted more freely and anonymous access-to-subprogram types are introduced. See subclauses 3.3, 3.6, 3.10, and 8.5.1.

- The control of structure and visibility has been enhanced to permit mutually dependent references between units and finer control over access from the private part of a package. See subclauses 3.10.1 and 10.1.2. In addition, limited types have been made more useful by the provision of aggregates, constants, and constructor functions. See subclauses 4.3, 6.5, and 7.5.
- The predefined environment has been extended to include additional time and calendar operations, improved string handling, a comprehensive container library, file and directory management, and access to environment variables. See subclauses 9.6.1, A.4, A.16, A.17, and A.18.
- Two of the Specialized Needs Annexes have been considerably enhanced:
  - The Real-Time Systems Annex now includes the Ravenscar profile for high-integrity systems, further dispatching policies such as Round Robin and Earliest Deadline First, support for timing events, and support for control of CPU time utilization. See subclauses D.2, D.13, D.14, and D.15.
  - The Numerics Annex now includes support for real and complex vectors and matrices as previously defined in ISO/IEC 13813:1997 plus further basic operations for linear algebra. See subclause G.3.
- The overall reliability of the language has been enhanced by a number of improvements. These include new syntax which detects accidental overloading, as well as pragmas for making assertions and giving better control over the suppression of checks. See subclauses 6.1, 11.4.2, and 11.5.

In addition, this third edition makes enhancements to address two important issues, namely, the particular problems of multiprocessor architectures, and the need to further increase the capabilities regarding assertions for correctness. It also makes additional changes and additions that improve the capability of the language and the reliability of programs written in the language.

The following significant changes with respect to the 1995 edition as amended by Amendment 1 are incorporated:

- New syntax (the aspect specification) is introduced to enable properties to be specified for various entities in a more structured manner than through pragmas. See subclause 13.1.1.
- The concept of assertions introduced in the 2005 edition is extended with the ability to specify preconditions and postconditions for subprograms, and invariants for private types. The concept of constraints in defining subtypes is supplemented with subtype predicates that enable subsets to be specified other than as simple ranges. These properties are all indicated using aspect specifications. See subclauses 3.2.4, 6.1.1, and 7.3.2.
- New forms of expressions are introduced. These are if expressions, case expressions, quantified expressions, and expression functions. As well as being useful for programming in general by avoiding the introduction of unnecessary assignments, they are especially valuable in conditions and invariants since they avoid the need to introduce auxiliary functions. See subclauses 4.5.7, 4.5.8, and 6.8. Membership tests are also made more flexible. See subclauses 4.4 and 4.5.2.
- A number of changes are made to subprogram parameters. Functions may now have parameters of all modes. In order to mitigate consequent (and indeed existing) problems of inadvertent order dependence, rules are introduced to reduce aliasing. A parameter may now be explicitly marked as aliased and the type of a parameter may be incomplete in certain circumstances. See subclauses 3.10.1, 6.1, and 6.4.1.
- The use of access types is now more flexible. The rules for accessibility and certain conversions are improved. See subclauses 3.10.2, 4.5.2, 4.6, and 8.6. Furthermore, better control of storage pools is provided. See subclause 13.11.4.

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

- The Real-Time Systems Annex now includes facilities for defining domains of processors and assigning tasks to them. Improvements are made to scheduling and budgeting facilities. See subclauses D.10.1, D.14, and D.16.
- A number of important improvements are made to the standard library. These include packages for conversions between strings and UTF encodings, and classification functions for wide and wide wide characters. Internationalization is catered for by a package giving locale information. See subclauses A.3, A.4.11, and A.19. The container library is extended to include bounded forms of the existing containers and new containers for indefinite objects, multiway trees, and queues. See subclause A.18.
- Finally, certain features are added primarily to ease the use of containers, such as the ability to iterate over all elements in a container without having to encode the iteration. These can also be used for iteration over arrays, and within quantified expressions. See subclauses 4.1.5, 4.1.6, 5.5.1, and 5.5.2.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

Instructions for Comment Submission

Informal comments on this International Standard may be sent via e-mail to **ada-comment@ada-auth.org**. If appropriate, the Project Editor will initiate the defect correction procedure.

Comments should use the following format:

> **!topic** *Title summarizing comment*
> **!reference** Ada 2012 RM*ss.ss(pp)*
> **!from** *Author Name yy-mm-dd*
> **!keywords** *keywords related to topic*
> **!discussion**
>
> *text of discussion*

where *ss.ss* is the clause or subclause number, *pp* is the paragraph number where applicable, and *yy-mm-dd* is the date the comment was sent. The date is optional, as is the **!keywords** line.

Please use a descriptive "Subject" in your e-mail message, and limit each message to a single comment.

When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [ ] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example:

> **!topic** [c]{C}haracter
> **!topic** it[']s meaning is not defined

Formal requests for interpretations and for reporting defects in this International Standard may be made in accordance with the ISO/IEC JTC 1 Directives and the ISO/IEC JTC 1/SC 22 policy for interpretations. National Bodies may submit a Defect Report to ISO/IEC JTC 1/SC 22 for resolution under the JTC 1 procedures. A response will be provided and, if appropriate, a Technical Corrigendum will be issued in accordance with the procedures.

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

INTERNATIONAL STANDARD **ISO/IEC 8652:2012(E)**

Information technology — Programming Languages — Ada

1 General

1.1 Scope

This International Standard 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.

Ada is a programming language designed to support the construction of long-lived, highly reliable software systems. The language includes facilities to define packages of related types, objects, and operations. The packages may be parameterized and the types may be extended to support the construction of libraries of reusable, adaptable software components. The operations may be implemented as subprograms using conventional sequential control structures, or as entries that include synchronization of concurrent threads of control as part of their invocation. Ada supports object-oriented programming by providing classes and interfaces, inheritance, polymorphism of variables and methods, and generic units. The language treats modularity in the physical sense as well, with a facility to support separate compilation.

The language provides rich support for real-time, concurrent programming, and includes facilities for multicore and multiprocessor programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, and random number generation, and definition and use of containers.

1.1.1 Extent

This International Standard specifies:

- The form of a program written in Ada;
- The effect of translating and executing such a program;
- The manner in which program units may be combined to form Ada programs;
- The language-defined library units that a conforming implementation is required to supply;
- The permissible variations within the standard, and the manner in which they are to be documented;

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

- Those violations of the standard 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 standard that a conforming implementation is not required to detect.

This International Standard 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.

1.1.2 Structure

This International Standard contains thirteen clauses, fifteen annexes, and an index.

The *core* of the Ada language consists of:

- Clauses 1 through 13
- Annex A, “Predefined Language Environment”
- Annex B, “Interface to Other Languages”
- Annex J, “Obsolescent Features”

The following *Specialized Needs Annexes* 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 core language and the Specialized Needs Annexes are normative, except that the material in each of the items listed below is informative:

- Text under a NOTES or Examples heading.
- Each subclause whose title starts with the word “Example” or “Examples”.

All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Specialized Needs Annexes.

The following Annexes are informative:

- Annex K, “Language-Defined Aspects and Attributes”
- Annex L, “Language-Defined Pragmas”
- Annex M, “Summary of Documentation Requirements”
- Annex N, “Glossary”

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

- Annex P, “Syntax Summary”
- Annex Q, “Language-Defined Entities”

Each section 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).

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 1.1.5).

Erroneous Execution

Situations that result in erroneous execution (see 1.1.5).

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. The word “should” is used to indicate that the advice is a recommendation, not a requirement. It is implementation defined whether or not a given recommendation is obeyed.

NOTES
1 Notes emphasize consequences of the rules described in the (sub)clause or elsewhere. This material is informative.

Examples

Examples illustrate the possible forms of the constructs described. This material is informative.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

1.1.3 Conformity of an Implementation with the Standard

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 International Standard;
- Supply all language-defined library units required by this International Standard;
- Contain no variations except those explicitly permitted by this International Standard, 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 International Standard.

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_statements (see 13.8); which code_statements 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 10.2) 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 International Standard 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 International Standard for the semantics of the given program.

An implementation that conforms to this Standard shall support each capability required by the core language as specified. In addition, an implementation that conforms to this Standard may conform to one or more Specialized Needs Annexes (or to none). Conformance to a Specialized Needs Annex means that each capability required by the Annex is provided as specified.

An implementation conforming to this International Standard 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.

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

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

documentation is not required for unspecified situations. The implementation-defined characteristics are summarized in M.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.

NOTES
2 The above requirements imply that an implementation conforming to this Standard may support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities.

1.1.4 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 sans-serif 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 unless it occurs immediately after an opening curly bracket, in which case it stands for itself:

  constraint ::= scalar_constraint | composite_constraint
  discrete_choice_list ::= discrete_choice {| discrete_choice}

- 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 International Standard (see 2.1) belong to the same range. For

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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 International Standard mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using simple upper case mapping, as defined by documents referenced in the note in Clause 1 of ISO/IEC 10646:2011.

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".

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 run-time 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.

NOTES

3 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;
```

4 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.

1.1.5 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 International Standard, 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

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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 need not 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 need not detect such errors 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 International Standard; 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.

1.2 Normative References

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. 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/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*.

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

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

ISO/IEC 3166-1:2006, *Codes for the representation of names of countries and their subdivisions — Part 1: Country Codes*.

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

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

ISO/IEC 8859-1:1998, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*.

ISO/IEC 9899:2011, *Information technology — Programming languages — C*.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

ISO/IEC 10646:2011, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

ISO/IEC 14882:2011, *Information technology — Programming languages — C++*.

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

1.3 Terms and Definitions

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

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

2 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.

2.1 Character Set

The character repertoire for the text of an Ada program consists of the entire coding space described by the ISO/IEC 10646:2011 Universal Multiple-Octet Coded Character Set. This coding space is organized in *planes*, each plane comprising 65536 characters.

Syntax

A character is defined by this International Standard for each cell in the coding space described by ISO/IEC 10646:2011, regardless of whether or not ISO/IEC 10646:2011 allocates a character to that cell.

Static Semantics

The coded representation for characters is implementation defined (it need not be a representation defined within ISO/IEC 10646:2011). 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 KC (as defined by Clause 21 of ISO/IEC 10646:2011) is implementation defined.

The description of the language definition in this International Standard uses the character properties General Category, Simple Uppercase Mapping, Uppercase Mapping, and Special Case Condition of the documents referenced by the note in Clause 1 of ISO/IEC 10646:2011. 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”.

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”.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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 10646:2011):

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

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 2.2); 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.

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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).

NOTES
1 The characters in categories other_control, other_private_use, and other_surrogate are only allowed in comments.

2.2 Lexical Elements, Separators, and Delimiters

Static Semantics

The text of a program consists of the texts of one or more compilations. 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 compilations, excluding comments.

The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than the character whose code 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:

& ' ( ) * + , – . / : ; < = > |

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”)

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

delimiter	name
>=	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.

2.3 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.

Static Semantics

Two identifiers 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 the note in Clause 1 of ISO/IEC 10646:2011.

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 identifiers, to accommodate local conventions.

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

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

Examples

Examples of identifiers:

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

2.4 Numeric Literals

There are two kinds of numeric_literals, *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

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

2.4.1 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.

Examples

Examples of decimal literals:

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

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

2.4.2 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

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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.

Examples

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
```

2.5 Character Literals

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

Syntax

character_literal ::= 'graphic_character'

NOTES
4 A character_literal is an enumeration literal of a character type. See 3.5.2.

Examples

Examples of character literals:

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

2.6 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 6.1), values of a string type (see 4.2), and array subaggregates (see 4.3.3).

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.

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

NOTES
5 An end of line cannot appear in a string_literal.

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

Examples

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) = "
```

2.7 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.

Examples

Examples of comments:

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

end;    -- processing of Line is complete

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

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

2.8 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.

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.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

Pragmas are only allowed at the following places in a program:

- After a semicolon delimiter, but not within a formal_part or discriminant_part.
- 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.

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_items.

Syntax

The forms of List, Page, and Optimize pragmas are as follows:

pragma List(identifier);

pragma Page;

pragma Optimize(identifier);

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

Other pragmas are defined throughout this International Standard, and are summarized in Annex L.

Static Semantics

A pragma List takes one of the identifiers 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 identifiers 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.

Examples

Examples of pragmas:

```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Pure(Rational_Numbers); -- set categorization for package
pragma Assert(Exists(File_Name),
              Message => "Nonexistent file"); -- assert file exists
```

2.9 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
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

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

NOTES
7 The reserved words appear in **lower case boldface** in this International Standard, except when used in the designator of an attribute (see 4.1.4). 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 may be written in whatever typeface is desired and available.

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

3 Declarations and Types

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

3.1 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 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
  | 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 loop_parameter_specification; an iterator_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 8.5)).

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 8.2). 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 8.3). 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 8.6). The declaration is said to *declare* the name, the view, and in most cases, the entity itself.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

As an alternative to an identifier, an enumeration literal can be declared with a character_literal as its name (see 3.5.1), and a function can be declared with an operator_symbol as its name (see 6.1).

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 identifiers, character_literals, and operator_symbols. 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 International Standard for each construct that has a run-time effect.

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

3.2 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 (see NOTES below), reflecting the similarity of their values and primitive operations. Most categories of types form *classes* of types. *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

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

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

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

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

index_constraints, and 3.7.1 for discriminant_constraints. The set of possible values for an object of an access type can also be subjected to a condition that excludes the null value (see 3.10).

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.

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).

NOTES

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

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.

3.2.1 Type Declarations

A type_declaration declares a type and its first subtype.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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
  | 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 3.7, "Discriminants"). 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 International Standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier. For a named type whose first subtype is T, this International Standard sometimes refers to the type of T as simply "the type T".

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 4.5, "Operators and Expression Evaluation".

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.

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

Examples

Examples of type definitions:

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

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;
```

NOTES
3 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 3.2.2). Although names do not directly denote types, a phrase like "the type Column" is sometimes used in this International Standard 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 3.3.1; its type is anonymous — it has no nameable subtypes.

3.2.2 Subtype Declarations

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

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.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

NOTES
4 A scalar_constraint may be applied to a subtype of an appropriate scalar type (see 3.5, 3.5.9, and J.3), even if the subtype is already constrained. On the other hand, a composite_constraint may be applied to a composite subtype (or an access-to-composite subtype) only if the composite subtype is unconstrained (see 3.6.1 and 3.7.1).

Examples

Examples of subtype declarations:

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

3.2.3 Classification of Operations

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

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

- The predefined operators of the type (see 4.5);
- For a derived type, the inherited (see 3.4) user-defined subprograms;
- For an enumeration type, the enumeration literals (which are considered parameterless functions — see 3.5.1);
- 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 6.6);
- 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 8.3) other implicitly declared primitive subprograms of the type.

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

3.2.4 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_specifications are found in Clause 13 (13.1 and 13.1.1 respectively).

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:

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

- For a (first) subtype defined by a derived type declaration, the predicates of the parent subtype and the progenitor subtypes apply.
- For a subtype created by a subtype_indication, the predicate of the subtype denoted by the subtype_mark applies.

The *predicate* of a subtype consists of all predicate specifications that apply, and-ed together; if no predicate specifications apply, the predicate is True (in particular, the predicate of a base subtype is True).

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 11.4.2) 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 11.4.2) 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;
- If a subtype is defined by a derived type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if predicate checks are enabled for at least one of the parent subtype and the 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.

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 simple_expression is the current instance, and whose membership_choice_list meets the requirements for a static membership test (see 4.9);
- a case_expression whose *selecting*_expression is the current instance, and whose *dependent*_expressions 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 logical operator, 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.

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

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 12.3), these rules apply also in the private part of an instance of a generic unit.

Dynamic Semantics

If predicate checks are enabled for a given subtype, then:

> On every subtype conversion, the predicate of the target subtype is evaluated, and a check is performed that the predicate is True. This includes all parameter passing, except for certain parameters passed by reference, which are covered by the following rule: After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by reference, the predicate of the subtype of the actual is evaluated, and a check is performed that the predicate is True. For an object created by an object_declaration with no explicit initialization expression, or by an uninitialized allocator, if any subcomponents have default_expressions, the predicate of the nominal subtype of the created object is evaluated, and a check is performed that the predicate is True. Assertions.Assertion_Error is raised if any of these checks fail.

A value *satisfies* a predicate if the predicate is True for that value.

If any of the above Legality Rules is violated in an instance of a generic unit, Program_Error is raised at the point of the violation.

NOTES

5 A predicate specification does not cause a subtype to be considered constrained.

6 A Static_Predicate, like a constraint, always remains True for all objects of the subtype, except in the case of uninitialized variables and other invalid values. A Dynamic_Predicate, on the other hand, is checked as specified above, but can become False at other times. For example, the predicate of a record subtype is not checked when a subcomponent is modified.

3.3 Objects and Named Numbers

Objects are created at run time and contain a value of a given type. An object can be created and initialized as part of elaborating a declaration, evaluating an allocator, aggregate, or function_call, or passing a parameter by copy. Prior to reclaiming the storage for an object, it is finalized if necessary (see 7.6.1).

Static Semantics

All of the following are objects:

- the entity declared by an object_declaration;
- a formal parameter of a subprogram, entry, or generic subprogram;
- a generic formal object;
- a loop parameter;
- a choice parameter of an exception_handler;
- an entry index of an entry_body;
- the result of dereferencing an access-to-object value (see 4.1);
- the return object of a function;
- the result of evaluating an aggregate;

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

- a qualified_expression whose operand denotes an object;
- a component, slice, or view conversion of another object.

An object is either a *constant* object or a *variable* object. Similarly, a view of an object is either a *constant* or a *variable*. All views of a constant elementary object are constant. All views of a constant composite object are constant, except for parts that are of controlled or immutably limited types; variable views of those parts and their subcomponents may exist. In this sense, objects of controlled and immutably limited types are *inherently mutable*. A constant view of an object cannot be used to modify its value. The terms constant and variable by themselves refer to constant and variable views of objects.

The value of an object is *read* when the value of any part of the object is evaluated, or when the value of an enclosing object is evaluated. The value of a variable is *updated* when an assignment is performed to any part of the variable, or when an assignment is performed to an enclosing object.

Whether a view of an object is constant or variable is determined by the definition of the view. The following (and no others) represent constants:

- an object declared by an object_declaration with the reserved word **constant**;
- a formal parameter or generic formal object of mode **in**;
- a discriminant;
- a loop parameter unless specified to be a variable for a generalized loop (see 5.5.2);
- a choice parameter or entry index;
- the dereference of an access-to-constant value;
- the return object declared by an extended_return_statement with the reserved word **constant**;
- the object denoted by a function_call or an aggregate;
- the result of evaluating a qualified_expression;
- within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit;
- a selected_component, indexed_component, slice, or view conversion of a constant.

At the place where a view of an object is defined, a *nominal subtype* is associated with the view. The object's *actual subtype* (that is, its subtype) can be more restrictive than the nominal subtype of the view; it always is if the nominal subtype is an *indefinite subtype*. A subtype is an indefinite subtype if it is an unconstrained array subtype, or if it has unknown discriminants or unconstrained discriminants without defaults (see 3.7); otherwise, the subtype is a *definite* subtype (all elementary subtypes are definite subtypes). A class-wide subtype is defined to have unknown discriminants, and is therefore an indefinite subtype. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary (see 3.3.1). A component cannot have an indefinite nominal subtype.

A view of a composite object is *known to be constrained* if:

- its nominal subtype is constrained, and is not an untagged partial view; or
- its nominal subtype is indefinite; or
- its type is immutably limited (see 7.5); or
- it is part of a stand-alone constant (including a generic formal object of mode **in**); or
- it is part of a formal parameter of mode **in**; or
- it is part of the object denoted by a function_call or aggregate; or
- it is part of a constant return object of an extended_return_statement; or

© ISO/IEC 2012 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012

- it is a dereference of a pool-specific access type, and there is no ancestor of its type that has a constrained partial view.

For the purposes of determining within a generic body whether an object is known to be constrained:

- if a subtype is a descendant of an untagged generic formal private or derived type, and the subtype is not an unconstrained array subtype, it is not considered indefinite and is considered to have a constrained partial view;
- if a subtype is a descendant of a formal access type, it is not considered pool-specific.

A *named number* provides a name for a numeric value known at compile time. It is declared by a number_declaration.

NOTES

7 A constant cannot be the target of an assignment operation, nor be passed as an **in out** or **out** parameter, between its initialization and finalization, if any.

8 The value of a constant object cannot be changed after its initialization, except in some cases where the object has a controlled or immutably limited part (see 7.5, 7.6, and 13.9.1).

9 The nominal and actual subtypes of an elementary object are always the same. For a discriminated or array object, if the nominal subtype is constrained, then so is the actual subtype.

3.3.1 Object Declarations

An object_declaration declares a *stand-alone* object with a given nominal subtype and, optionally, an explicit initial value given by an initialization expression. For an array, access, task, or protected object, the object_declaration may include the definition of the (anonymous) type of the object.

Syntax

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

defining_identifier_list ::=
  defining_identifier {, defining_identifier}
```

Name Resolution Rules

For an object_declaration with an expression following the compound delimiter :=, the type expected for the expression is that of the object. This expression is called the *initialization expression*.

Legality Rules

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

Static Semantics

An object_declaration with the reserved word **constant** declares a constant object. If it has an initialization expression, then it is called a *full constant declaration*. Otherwise, it is called a *deferred constant declaration*. The rules for deferred constant declarations are given in subclause 7.4. The rules for full constant declarations are given in this subclause.

© ISO/IEC 2012 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 8652:2012