

The Java™ Language Specification

Java SE 7 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

Copyright © 1997, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Duke logo™ designed by Joe Palrang.

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Oracle hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Oracle's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification.

2. License for the Distribution of Compliant Implementations. Oracle also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope.

No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Oracle in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Oracle's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the

Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b. With respect to any patent claims owned by Oracle and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Oracle that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c. Also with respect to any patent claims owned by Oracle and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Oracle that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Oracle's source code or binary code materials nor, except with an appropriate and separate license from Oracle, includes any of Oracle's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Oracle which corresponds to the Specification and that was available either (i) from Oracle 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Oracle if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ORACLE OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL,

INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION, EVEN IF ORACLE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Oracle and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

If you provide Oracle with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Table of Contents

1 Introduction 1

- 1.1 Example Programs 5
- 1.2 Notation 6
- 1.3 Relationship to Predefined Classes and Interfaces 6
- 1.4 References 7

2 Grammars 9

- 2.1 Context-Free Grammars 9
- 2.2 The Lexical Grammar 9
- 2.3 The Syntactic Grammar 10
- 2.4 Grammar Notation 10

3 Lexical Structure 13

- 3.1 Unicode 13
- 3.2 Lexical Translations 14
- 3.3 Unicode Escapes 15
- 3.4 Line Terminators 16
- 3.5 Input Elements and Tokens 17
- 3.6 White Space 18
- 3.7 Comments 18
- 3.8 Identifiers 20
- 3.9 Keywords 20
- 3.10 Literals 21
 - 3.10.1 Integer Literals 21
 - 3.10.2 Floating-Point Literals 24
 - 3.10.3 Boolean Literals 26
 - 3.10.4 Character Literals 26
 - 3.10.5 String Literals 27
 - 3.10.6 Escape Sequences for Character and String Literals 28
 - 3.10.7 The Null Literal 29
- 3.11 Separators 29
- 3.12 Operators 29

4 Types, Values, and Variables 31

- 4.1 The Kinds of Types and Values 32
- 4.2 Primitive Types and Values 32
 - 4.2.1 Integral Types and Values 33

4.2.2	Integer Operations	34
4.2.3	Floating-Point Types, Formats, and Values	35
4.2.4	Floating-Point Operations	38
4.2.5	The <code>boolean</code> Type and <code>boolean</code> Values	40
4.3	Reference Types and Values	41
4.3.1	Objects	42
4.3.2	The Class <code>Object</code>	43
4.3.3	The Class <code>String</code>	44
4.3.4	When Reference Types Are the Same	44
4.4	Type Variables	44
4.5	Parameterized Types	45
4.5.1	Type Arguments and Wildcards	46
4.5.2	Members and Constructors of Parameterized Types	48
4.6	Type Erasure	48
4.7	Reifiable Types	49
4.8	Raw Types	50
4.9	Intersection Types	51
4.10	Subtyping	51
4.10.1	Subtyping among Primitive Types	52
4.10.2	Subtyping among Class and Interface Types	52
4.10.3	Subtyping among Array Types	53
4.11	Where Types Are Used	53
4.12	Variables	54
4.12.1	Variables of Primitive Type	54
4.12.2	Variables of Reference Type	54
4.12.3	Kinds of Variables	54
4.12.4	<code>final</code> Variables	56
4.12.5	Initial Values of Variables	57
4.12.6	Types, Classes, and Interfaces	57

5 Conversions and Promotions 59

5.1	Kinds of Conversion	61
5.1.1	Identity Conversions	61
5.1.2	Widening Primitive Conversion	61
5.1.3	Narrowing Primitive Conversions	62
5.1.4	Widening and Narrowing Primitive Conversions	63
5.1.5	Widening Reference Conversions	64
5.1.6	Narrowing Reference Conversions	64
5.1.7	Boxing Conversion	65
5.1.8	Unboxing Conversion	66
5.1.9	Unchecked Conversion	67
5.1.10	Capture Conversion	67
5.1.11	String Conversions	68
5.1.12	Forbidden Conversions	69
5.1.13	Value Set Conversion	69
5.2	Assignment Conversion	70
5.3	Method Invocation Conversion	72

- 5.4 String Conversion 73
- 5.5 Casting Conversion 73
 - 5.5.1 Reference Type Casting 74
 - 5.5.2 Checked Casts and Unchecked Casts 76
 - 5.5.3 Checked Casts at Run-time 77
- 5.6 Numeric Promotions 78
 - 5.6.1 Unary Numeric Promotion 78
 - 5.6.2 Binary Numeric Promotion 79

6 Names 81

- 6.1 Declarations 82
- 6.2 Names and Identifiers 83
- 6.3 Scope of a Declaration 84
- 6.4 Shadowing and Obscuring 86
 - 6.4.1 Shadowing 86
 - 6.4.2 Obscuring 87
- 6.5 Determining the Meaning of a Name 88
 - 6.5.1 Syntactic Classification of a Name According to Context 89
 - 6.5.2 Reclassification of Contextually Ambiguous Names 91
 - 6.5.3 Meaning of Package Names 92
 - 6.5.3.1 Simple Package Names 92
 - 6.5.3.2 Qualified Package Names 93
 - 6.5.4 Meaning of *PackageOrTypeNames* 93
 - 6.5.4.1 Simple *PackageOrTypeNames* 93
 - 6.5.4.2 Qualified *PackageOrTypeNames* 93
 - 6.5.5 Meaning of Type Names 93
 - 6.5.5.1 Simple Type Names 93
 - 6.5.5.2 Qualified Type Names 93
 - 6.5.6 Meaning of Expression Names 94
 - 6.5.6.1 Simple Expression Names 94
 - 6.5.6.2 Qualified Expression Names 94
 - 6.5.7 Meaning of Method Names 96
 - 6.5.7.1 Simple Method Names 96
 - 6.5.7.2 Qualified Method Names 96
- 6.6 Access Control 97
 - 6.6.1 Determining Accessibility 97
 - 6.6.2 Details on protected Access 98
 - 6.6.2.1 Access to a **protected** Member 98
 - 6.6.2.2 Qualified Access to a **protected** Constructor 98
- 6.7 Fully Qualified Names and Canonical Names 99

7 Packages 101

- 7.1 Package Members 101
- 7.2 Host Support for Packages 102
- 7.3 Compilation Units 102
- 7.4 Package Declarations 103
 - 7.4.1 Named Packages 104

- 7.4.2 Unnamed Packages 104
- 7.4.3 Observability of a Package 104
- 7.5 Import Declarations 105
 - 7.5.1 Single-Type-Import Declaration 105
 - 7.5.2 Type-Import-on-Demand Declaration 106
 - 7.5.3 Single Static Import Declaration 107
 - 7.5.4 Static-Import-on-Demand Declaration 107
- 7.6 Top Level Type Declarations 108

8 Classes 111

- 8.1 Class Declaration 113
 - 8.1.1 Class Modifiers 113
 - 8.1.1.1 `abstract` Classes 114
 - 8.1.1.2 `final` Classes 115
 - 8.1.1.3 `strictfp` Classes 115
 - 8.1.2 Generic Classes and Type Parameters 115
 - 8.1.3 Inner Classes and Enclosing Instances 116
 - 8.1.4 Superclasses and Subclasses 118
 - 8.1.5 Superinterfaces 119
 - 8.1.6 Class Body and Member Declarations 120
- 8.2 Class Members 121
- 8.3 Field Declarations 122
 - 8.3.1 Field Modifiers 124
 - 8.3.1.1 `static` Fields 124
 - 8.3.1.2 `final` Fields 124
 - 8.3.1.3 `transient` Fields 125
 - 8.3.1.4 `volatile` Fields 125
 - 8.3.2 Initialization of Fields 125
 - 8.3.2.1 Initializers for Class Variables 125
 - 8.3.2.2 Initializers for Instance Variables 125
 - 8.3.2.3 Restrictions on the use of Fields during Initialization 126
- 8.4 Method Declarations 126
 - 8.4.1 Formal Parameters 127
 - 8.4.2 Method Signature 129
 - 8.4.3 Method Modifiers 130
 - 8.4.3.1 `abstract` Methods 131
 - 8.4.3.2 `static` Methods 131
 - 8.4.3.3 `final` Methods 131
 - 8.4.3.4 `native` Methods 132
 - 8.4.3.5 `strictfp` Methods 132
 - 8.4.3.6 `synchronized` Methods 132
 - 8.4.4 Generic Methods 132
 - 8.4.5 Method Return Type 132
 - 8.4.6 Method Throws 133
 - 8.4.7 Method Body 134
 - 8.4.8 Inheritance, Overriding, and Hiding 134

	8.4.8.1	Overriding (by Instance Methods)	135
	8.4.8.2	Hiding (by Class Methods)	135
	8.4.8.3	Requirements in Overriding and Hiding	136
	8.4.8.4	Inheriting Methods with Override-Equivalent Signatures	136
	8.4.9	Overloading	137
8.5		Member Type Declarations	138
	8.5.1	Access Modifiers	138
	8.5.2	Static Member Type Declarations	139
8.6		Instance Initializers	139
8.7		Static Initializers	139
8.8		Constructor Declarations	140
	8.8.1	Formal Parameters and Type Parameters	141
	8.8.2	Constructor Signature	141
	8.8.3	Constructor Modifiers	141
	8.8.4	Generic Constructors	141
	8.8.5	Constructor Throws	142
	8.8.6	The Type of a Constructor	142
	8.8.7	Constructor Body	142
	8.8.7.1	Explicit Constructor Invocations	143
	8.8.8	Constructor Overloading	145
	8.8.9	Default Constructor	145
	8.8.10	Preventing Instantiation of a Class	146
8.9		Enums	146
	8.9.1	Enum Constants	146
	8.9.2	Enum Body and Member Declarations	148

9 Interfaces 151

9.1		Interface Declarations	152
	9.1.1	Interface Modifiers	152
	9.1.1.1	<code>abstract</code> Interfaces	153
	9.1.1.2	<code>strictfp</code> Interfaces	153
	9.1.2	Generic Interfaces and Type Parameters	153
	9.1.3	Superinterfaces and Subinterfaces	154
	9.1.4	Interface Body and Member Declarations	155
9.2		Interface Members	155
9.3		Field (Constant) Declarations	156
	9.3.1	Initialization of Fields in Interfaces	157
9.4		Abstract Method Declarations	158
	9.4.1	Inheritance and Overriding	159
	9.4.2	Overloading	160
9.5		Member Type Declarations	160
9.6		Annotation Types	161
	9.6.1	Annotation Type Elements	162
	9.6.2	Defaults for Annotation Type Elements	163
	9.6.3	Predefined Annotation Types	163
	9.6.3.1	<code>Target</code>	163

	9.6.3.2	Retention	163
	9.6.3.3	Inherited	164
	9.6.3.4	Override	164
	9.6.3.5	SuppressWarnings	164
	9.6.3.6	Deprecated	165
9.7	Annotations		165
	9.7.1	Normal Annotations	166
	9.7.2	Marker Annotations	168
	9.7.3	Single-Element Annotations	168

10 Arrays 171

10.1	Array Types	172
10.2	Array Variables	172
10.3	Array Creation	173
10.4	Array Access	173
10.5	Array Store Exception	174
10.6	Array Initializers	174
10.7	Array Members	175
10.8	Class Objects for Arrays	176
10.9	An Array of Characters is Not a String	176

11 Exceptions 177

11.1	The Kinds and Causes of Exceptions	178
	11.1.1 The Kinds of Exceptions	178
	11.1.2 The Causes of Exceptions	179
	11.1.3 Asynchronous Exceptions	179
11.2	Compile-Time Checking of Exceptions	180
	11.2.1 Exception Analysis of Expressions	180
	11.2.2 Exception Analysis of Statements	181
	11.2.3 Exception Checking	182
11.3	Run-Time Handling of an Exception	182

12 Execution 185

12.1	Java virtual machine Start-Up	185
	12.1.1 Load the Class <code>Test</code>	186
	12.1.2 Link <code>Test</code> : Verify, Prepare, (Optionally) Resolve	186
	12.1.3 Initialize <code>Test</code> : Execute Initializers	187
	12.1.4 Invoke <code>Test.main</code>	188
12.2	Loading of Classes and Interfaces	188
	12.2.1 The Loading Process	189
12.3	Linking of Classes and Interfaces	189
	12.3.1 Verification of the Binary Representation	190
	12.3.2 Preparation of a Class or Interface Type	190
	12.3.3 Resolution of Symbolic References	191
12.4	Initialization of Classes and Interfaces	192
	12.4.1 When Initialization Occurs	192

- 12.4.2 Detailed Initialization Procedure 193
- 12.5 Creation of New Class Instances 195
- 12.6 Finalization of Class Instances 197
 - 12.6.1 Implementing Finalization 198
 - 12.6.1.1 Interaction with the Memory Model 198
 - 12.6.2 Finalizer Invocations are Not Ordered 200
- 12.7 Unloading of Classes and Interfaces 200
- 12.8 Program Exit 200

13 Binary Compatibility 201

- 13.1 The Form of a Binary 202
- 13.2 What Binary Compatibility Is and Is Not 207
- 13.3 Evolution of Packages 207
- 13.4 Evolution of Classes 208
 - 13.4.1 `abstract` Classes 208
 - 13.4.2 `final` Classes 208
 - 13.4.3 `public` Classes 208
 - 13.4.4 Superclasses and Superinterfaces 209
 - 13.4.5 Class Type Parameters 209
 - 13.4.6 Class Body and Member Declarations 209
 - 13.4.7 Access to Members and Constructors 210
 - 13.4.8 Field Declarations 210
 - 13.4.9 `final` Fields and Constants 211
 - 13.4.10 `static` Fields 212
 - 13.4.11 `transient` Fields 212
 - 13.4.12 Method and Constructor Declarations 213
 - 13.4.13 Method and Constructor Type Parameters 213
 - 13.4.14 Method and Constructor Formal Parameters 214
 - 13.4.15 Method Result Type 214
 - 13.4.16 `abstract` Methods 215
 - 13.4.17 `final` Methods 215
 - 13.4.18 `native` Methods 215
 - 13.4.19 `static` Methods 216
 - 13.4.20 `synchronized` Methods 216
 - 13.4.21 Method and Constructor Throws 216
 - 13.4.22 Method and Constructor Body 216
 - 13.4.23 Method and Constructor Overloading 216
 - 13.4.24 Method Overriding 217
 - 13.4.25 Static Initializers 217
 - 13.4.26 Evolution of Enums 217
- 13.5 Evolution of Interfaces 217
 - 13.5.1 `public` Interfaces 217
 - 13.5.2 Superinterfaces 218
 - 13.5.3 The Interface Members 218
 - 13.5.4 Interface Type Parameters 218
 - 13.5.5 Field Declarations 218
 - 13.5.6 `abstract` Methods 218

13.5.7 Evolution of Annotation Types 219

14 Blocks and Statements 221

- 14.1 Normal and Abrupt Completion of Statements 221
- 14.2 Blocks 223
- 14.3 Local Class Declarations 223
- 14.4 Local Variable Declaration Statements 224
 - 14.4.1 Local Variable Declarators and Types 224
 - 14.4.2 Local Variable Names 225
 - 14.4.3 Execution of Local Variable Declarations 225
- 14.5 Statements 226
- 14.6 The Empty Statement 228
- 14.7 Labeled Statements 228
- 14.8 Expression Statements 229
- 14.9 The `if` Statement 230
 - 14.9.1 The `if-then` Statement 230
 - 14.9.2 The `if-then-else` Statement 230
- 14.10 The `assert` Statement 231
- 14.11 The `switch` Statement 232
- 14.12 The `while` Statement 235
 - 14.12.1 Abrupt Completion 235
- 14.13 The `do` Statement 236
 - 14.13.1 Abrupt Completion 237
- 14.14 The `for` Statement 237
 - 14.14.1 The basic `for` Statement 238
 - 14.14.1.1 Initialization of `for` statement 238
 - 14.14.1.2 Iteration of `for` statement 239
 - 14.14.1.3 Abrupt Completion of `for` statement 240
 - 14.14.2 The enhanced `for` statement 240
- 14.15 The `break` Statement 241
- 14.16 The `continue` Statement 242
- 14.17 The `return` Statement 243
- 14.18 The `throw` Statement 244
- 14.19 The `synchronized` Statement 246
- 14.20 The `try` statement 247
 - 14.20.1 Execution of `try-catch` 248
 - 14.20.2 Execution of `try-finally` and `try-catch-finally` 249
- 14.21 Unreachable Statements 250

15 Expressions 255

- 15.1 Evaluation, Denotation, and Result 255
- 15.2 Variables as Values 256
- 15.3 Type of an Expression 256
- 15.4 FP-strict Expressions 257
- 15.5 Expressions and Run-Time Checks 257
- 15.6 Normal and Abrupt Completion of Evaluation 259

- 15.7 Evaluation Order 261
 - 15.7.1 Evaluate Left-Hand Operand First 261
 - 15.7.2 Evaluate Operands before Operation 261
 - 15.7.3 Evaluation Respects Parentheses and Precedence 261
 - 15.7.4 Argument Lists are Evaluated Left-to-Right 262
 - 15.7.5 Evaluation Order for Other Expressions 262
- 15.8 Primary Expressions 262
 - 15.8.1 Lexical Literals 263
 - 15.8.2 Class Literals 264
 - 15.8.3 `this` 264
 - 15.8.4 Qualified `this` 265
 - 15.8.5 Parenthesized Expressions 265
- 15.9 Class Instance Creation Expressions 266
 - 15.9.1 Determining the Class being Instantiated 267
 - 15.9.2 Determining Enclosing Instances 268
 - 15.9.3 Choosing the Constructor and its Arguments 270
 - 15.9.4 Run-time Evaluation of Class Instance Creation Expressions 270
 - 15.9.5 Anonymous Class Declarations 271
 - 15.9.5.1 Anonymous Constructors 271
- 15.10 Array Creation Expressions 273
 - 15.10.1 Run-time Evaluation of Array Creation Expressions 274
- 15.11 Field Access Expressions 275
 - 15.11.1 Field Access Using a Primary 275
 - 15.11.2 Accessing Superclass Members using `super` 276
- 15.12 Method Invocation Expressions 277
 - 15.12.1 Compile-Time Step 1: Determine Class or Interface to Search 277
 - 15.12.2 Compile-Time Step 2: Determine Method Signature 279
 - 15.12.2.1 Identify Potentially Applicable Methods 280
 - 15.12.2.2 Phase 1: Identify Matching Arity Methods Applicable by Subtyping 281
 - 15.12.2.3 Phase 2: Identify Matching Arity Methods Applicable by Method Invocation Conversion 282
 - 15.12.2.4 Phase 3: Identify Applicable Variable Arity Methods 282
 - 15.12.2.5 Choosing the Most Specific Method 283
 - 15.12.2.6 Method Result and Throws Types 286
 - 15.12.2.7 Inferring Type Arguments Based on Actual Arguments 286
 - 15.12.2.8 Inferring Unresolved Type Arguments 291
 - 15.12.3 Compile-Time Step 3: Is the Chosen Method Appropriate? 293
 - 15.12.4 Runtime Evaluation of Method Invocation 294
 - 15.12.4.1 Compute Target Reference (If Necessary) 295
 - 15.12.4.2 Evaluate Arguments 296
 - 15.12.4.3 Check Accessibility of Type and Method 296
 - 15.12.4.4 Locate Method to Invoke 297
 - 15.12.4.5 Create Frame, Synchronize, Transfer Control 298

15.13	Array Access Expressions	299
15.13.1	Runtime Evaluation of Array Access	300
15.14	Postfix Expressions	301
15.14.1	Expression Names	301
15.14.2	Postfix Increment Operator ++	301
15.14.3	Postfix Decrement Operator --	302
15.15	Unary Operators	302
15.15.1	Prefix Increment Operator ++	303
15.15.2	Prefix Decrement Operator --	304
15.15.3	Unary Plus Operator +	304
15.15.4	Unary Minus Operator -	305
15.15.5	Bitwise Complement Operator ~	305
15.15.6	Logical Complement Operator !	306
15.16	Cast Expressions	306
15.17	Multiplicative Operators	307
15.17.1	Multiplication Operator *	307
15.17.2	Division Operator /	308
15.17.3	Remainder Operator %	310
15.18	Additive Operators	311
15.18.1	String Concatenation Operator +	311
15.18.2	Additive Operators (+ and -) for Numeric Types	312
15.19	Shift Operators	314
15.20	Relational Operators	315
15.20.1	Numerical Comparison Operators <, <=, >, and >=	315
15.20.2	Type Comparison Operator instanceof	316
15.21	Equality Operators	317
15.21.1	Numerical Equality Operators == and !=	317
15.21.2	Boolean Equality Operators == and !=	318
15.21.3	Reference Equality Operators == and !=	319
15.22	Bitwise and Logical Operators	319
15.22.1	Integer Bitwise Operators &, ^, and	320
15.22.2	Boolean Logical Operators &, ^, and	320
15.23	Conditional-And Operator &&	321
15.24	Conditional-Or Operator	321
15.25	Conditional Operator ? :	322
15.26	Assignment Operators	324
15.26.1	Simple Assignment Operator =	325
15.26.2	Compound Assignment Operators	327
15.27	Expression	329
15.28	Constant Expression	329

16 Definite Assignment 331

16.1	Definite Assignment and Expressions	333
16.1.1	Boolean Constant Expressions	333
16.1.2	The Boolean Operator &&	333
16.1.3	The Boolean Operator	333
16.1.4	The Boolean Operator !	334

16.1.5	The Boolean Operator <code>?</code> :	334
16.1.6	The Conditional Operator <code>?:</code> :	334
16.1.7	Other Expressions of Type <code>boolean</code>	335
16.1.8	Assignment Expressions	335
16.1.9	Operators <code>++</code> and <code>--</code>	335
16.1.10	Other Expressions	335
16.2	Definite Assignment and Statements	337
16.2.1	Empty Statements	337
16.2.2	Blocks	337
16.2.3	Local Class Declaration Statements	338
16.2.4	Local Variable Declaration Statements	338
16.2.5	Labeled Statements	338
16.2.6	Expression Statements	339
16.2.7	<code>if</code> Statements	339
16.2.8	<code>assert</code> Statements	339
16.2.9	<code>switch</code> Statements	339
16.2.10	<code>while</code> Statements	340
16.2.11	<code>do</code> Statements	341
16.2.12	<code>for</code> Statements	341
	16.2.12.1 Initialization Part	342
	16.2.12.2 Incrementation Part	342
16.2.13	<code>break</code> , <code>continue</code> , <code>return</code> , and <code>throw</code> Statements	343
16.2.14	<code>synchronized</code> Statements	343
16.2.15	<code>try</code> Statements	343
16.3	Definite Assignment and Parameters	345
16.4	Definite Assignment and Array Initializers	345
16.5	Definite Assignment and Enum Constants	345
16.6	Definite Assignment and Anonymous Classes	346
16.7	Definite Assignment and Member Types	346
16.8	Definite Assignment and Static Initializers	346
16.9	Definite Assignment, Constructors, and Instance Initializers	347

17 Threads and Locks 349

17.1	Synchronization	350
17.2	Wait Sets and Notification	350
	17.2.1 Wait	351
	17.2.2 Notification	352
	17.2.3 Interruptions	353
	17.2.4 Interactions of Waits, Notification, and Interruption	353
17.3	Sleep and Yield	354
17.4	Memory Model	354
	17.4.1 Shared Variables	355
	17.4.2 Actions	355
	17.4.3 Programs and Program Order	357
	17.4.4 Synchronization Order	357
	17.4.5 Happens-before Order	358
	17.4.6 Executions	360

17.4.7	Well-Formed Executions	360
17.4.8	Executions and Causality Requirements	361
17.4.9	Observable Behavior and Nonterminating Executions	362
17.5	final Field Semantics	364
17.5.1	Semantics of final Fields	365
17.5.2	Reading final Fields During Construction	366
17.5.3	Subsequent Modification of final Fields	366
17.5.4	Write-protected Fields	367
17.6	Word Tearing	367
17.7	Non-atomic Treatment of double and long	367

18 Syntax 369

Introduction

THE Java™ programming language is a general-purpose, concurrent, class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, the design has avoided including new and untested features.

The Java programming language is strongly typed. This specification clearly distinguishes between the compile-time errors that can and must be detected at compile time, and those that occur at run time. Compile time normally consists of translating programs into a machine-independent byte code representation. Run-time activities include loading and linking of the classes needed to execute a program, optional machine code generation and dynamic optimization of the program, and actual program execution.

The Java programming language is a relatively high-level language, in that details of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation (as in C's `free` or C++'s `delete`). High-performance garbage-collected implementations can have bounded pauses to support systems programming and real-time applications. The language does not include any unsafe constructs, such as array accesses without index checking, since such unsafe constructs would cause a program to behave in an unspecified way.

The Java programming language is normally compiled to the bytecoded instruction set and binary format defined in *The Java™ Virtual Machine Specification, Java SE 7 Edition*.

This specification is organized as follows:

Chapter 2 describes grammars and the notation used to present the lexical and syntactic grammars for the language.

Chapter 3 describes the lexical structure of the Java programming language, which is based on C and C++. The language is written in the Unicode character set. It supports the writing of Unicode characters on systems that support only ASCII.

Chapter 4 describes types, values, and variables. Types are subdivided into primitive types and reference types.

The primitive types are defined to be the same on all machines and in all implementations, and are various sizes of two's-complement integers, single- and double-precision IEEE 754 standard floating-point numbers, a `boolean` type, and a Unicode character `char` type. Values of the primitive types do not share state.

Reference types are the class types, the interface types, and the array types. The reference types are implemented by dynamically created objects that are either instances of classes or arrays. Many references to each object can exist. All objects (including arrays) support the methods of the class `Object`, which is the (single) root of the class hierarchy. A predefined `String` class supports Unicode character strings. Classes exist for wrapping primitive values inside of objects. In many cases, wrapping and unwrapping is performed automatically by the compiler (in which case, wrapping is called boxing, and unwrapping is called unboxing). Class and interface declarations may be generic, that is, they may be parameterized by other reference types. Such declarations may then be invoked with specific type arguments.

Variables are typed storage locations. A variable of a primitive type holds a value of that exact primitive type. A variable of a class type can hold a null reference or a reference to an object whose type is that class type or any subclass of that class type. A variable of an interface type can hold a null reference or a reference to an instance of any class that implements the interface. A variable of an array type can hold a null reference or a reference to an array. A variable of class type `Object` can hold a null reference or a reference to any object, whether class instance or array.

Chapter 5 describes conversions and numeric promotions. Conversions change the compile-time type and, sometimes, the value of an expression. These conversions include the boxing and unboxing conversions between primitive types and reference types. Numeric promotions are used to convert the operands of a numeric operator to a common type where an operation can be performed. There are no loopholes in the language; casts on reference types are checked at run time to ensure type safety.

Chapter 6 describes declarations and names, and how to determine what names mean (denote). The language does not require types or their members to be declared before they are used. Declaration order is significant only for local variables, local classes, and the order of initializers of fields in a class or interface.

The Java programming language provides control over the scope of names and supports limitations on external access to members of packages, classes, and interfaces. This helps in writing large programs by distinguishing the implementation of a type from its users and those who extend it. Recommended naming conventions that make for more readable programs are described here.

Chapter 7 describes the structure of a program, which is organized into packages similar to the modules of Modula. The members of a package are classes, interfaces, and subpackages. Packages are divided into compilation units. Compilation units contain type declarations and can import types from other packages to give them short names. Packages have names in a hierarchical name space, and the Internet domain name system can usually be used to form unique package names.

Chapter 8 describes classes. The members of classes are classes, interfaces, fields (variables) and methods. Class variables exist once per class. Class methods operate without reference to a specific object. Instance variables are dynamically created in objects that are instances of classes. Instance methods are invoked on instances of classes; such instances become the current object `this` during their execution, supporting the object-oriented programming style.

Classes support single implementation inheritance, in which the implementation of each class is derived from that of a single superclass, and ultimately from the class `Object`. Variables of a class type can reference an instance of that class or of any subclass of that class, allowing new types to be used with existing methods, polymorphically.

Classes support concurrent programming with synchronized methods. Methods declare the checked exceptions that can arise from their execution, which allows compile-time checking to ensure that exceptional conditions are handled. Objects can declare a `finalize` method that will be invoked before the objects are discarded by the garbage collector, allowing the objects to clean up their state.

For simplicity, the language has neither declaration "headers" separate from the implementation of a class nor separate type and class hierarchies.

A special form of classes, enums, support the definition of small sets of values and their manipulation in a type safe manner. Unlike enumerations in other languages, enums are objects and may have their own methods.

Chapter 9 describes interface types, which declare a set of abstract methods, member types, and constants. Classes that are otherwise unrelated can implement the same interface type. A variable of an interface type can contain a reference to any object that implements the interface. Multiple interface inheritance is supported.

Annotation types are specialized interfaces used to annotate declarations. Such annotations are not permitted to affect the semantics of programs in the Java programming language in any way. However, they provide useful input to various tools.

Chapter 10 describes arrays. Array accesses include bounds checking. Arrays are dynamically created objects and may be assigned to variables of type `Object`. The language supports arrays of arrays, rather than multidimensional arrays.

Chapter 11 describes exceptions, which are nonresuming and fully integrated with the language semantics and concurrency mechanisms. There are three kinds of exceptions: checked exceptions, run-time exceptions, and errors. The compiler ensures that checked exceptions are properly handled by requiring that a method or constructor can result in a checked exception only if the method or constructor declares it. This provides compile-time checking that exception handlers exist, and aids programming in the large. Most user-defined exceptions should be checked exceptions. Invalid operations in the program detected by the Java virtual machine result in run-time exceptions, such as `NullPointerException`. Errors result from failures detected by the Java virtual machine, such as `OutOfMemoryError`. Most simple programs do not try to handle errors.

Chapter 12 describes activities that occur during execution of a program. A program is normally stored as binary files representing compiled classes and interfaces. These binary files can be loaded into a Java virtual machine, linked to other classes and interfaces, and initialized.

After initialization, class methods and class variables may be used. Some classes may be instantiated to create new objects of the class type. Objects that are class instances also contain an instance of each superclass of the class, and object creation involves recursive creation of these superclass instances.

When an object is no longer referenced, it may be reclaimed by the garbage collector. If an object declares a finalizer, the finalizer is executed before the object is reclaimed to give the object a last chance to clean up resources that would not otherwise be released. When a class is no longer needed, it may be unloaded.

Chapter 13 describes binary compatibility, specifying the impact of changes to types on other types that use the changed types but have not been recompiled. These

considerations are of interest to developers of types that are to be widely distributed, in a continuing series of versions, often through the Internet. Good program development environments automatically recompile dependent code whenever a type is changed, so most programmers need not be concerned about these details.

Chapter 14 describes blocks and statements, which are based on C and C++. The language has no `goto` statement, but includes labeled `break` and `continue` statements. Unlike C, the Java programming language requires `boolean` (or `Boolean`) expressions in control-flow statements, and does not convert types to `boolean` implicitly (except through unboxing), in the hope of catching more errors at compile time. A `synchronized` statement provides basic object-level monitor locking. A `try` statement can include `catch` and `finally` clauses to protect against non-local control transfers.

Chapter 15 describes expressions. This document fully specifies the (apparent) order of evaluation of expressions, for increased determinism and portability. Overloaded methods and constructors are resolved at compile time by picking the most specific method or constructor from those which are applicable.

Chapter 16 describes the precise way in which the language ensures that local variables are definitely set before use. While all other variables are automatically initialized to a default value, the Java programming language does not automatically initialize local variables in order to avoid masking programming errors.

Chapter 17 describes the semantics of threads and locks, which are based on the monitor-based concurrency originally introduced with the Mesa programming language. The Java programming language specifies a memory model for shared-memory multiprocessors that supports high-performance implementations.

Chapter 18 presents a syntactic grammar for the language.

1.1 Example Programs

Most of the example programs given in the text are ready to be executed and are similar in form to:

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
        System.out.println();
    }
}
```

On a machine with Oracle's Java Development Kit installed, this class, stored in the file `Test.java`, can be compiled and executed by giving the commands:

```
javac Test.java
java Test Hello, world.
```

producing the output:

```
Hello, world.
```

1.2 Notation

Throughout this specification we refer to classes and interfaces drawn from the Java SE API. Whenever we refer to a class or interface which is not defined in an example in this specification using a single identifier *N*, the intended reference is to the class or interface named *N* in the package `java.lang`. We use the canonical name (§6.7) for classes or interfaces from packages other than `java.lang`.

Whenever we refer to *The Java™ Virtual Machine Specification* in this specification, we mean the Java SE 7 Edition.

1.3 Relationship to Predefined Classes and Interfaces

As noted above, this specification often refers to classes of the Java SE API. In particular, some classes have a special relationship with the Java programming language. Examples include classes such as `Object`, `Class`, `ClassLoader`, `String`, `Thread`, and the classes and interfaces in package `java.lang.reflect`, among others. The language definition constrains the behavior of these classes and interfaces, but this document does not provide a complete specification for them. The reader is referred to other parts of the Java SE platform Specification for such detailed API specifications.

Thus this document does not describe reflection in any detail. Many linguistic constructs have analogues in the reflection API, but these are generally not discussed here. So, for example, when we list the ways in which an object can be created, we generally do not include the ways in which the reflective API can accomplish this. Readers should be aware of these additional mechanisms even though they are not mentioned in this text.

1.4 References

- Apple Computer. *Dylan™ Reference Manual*. Apple Computer Inc., Cupertino, California. September 29, 1995.
- Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification*, X3J13 Document 88-002R, June 1988; appears as Chapter 28 of Steele, Guy. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990, ISBN 1-55558-041-6, 770-864.
- Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990, reprinted with corrections October 1992, ISBN 0-201-51459-1.
- Goldberg, Adele and Robson, David. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989, ISBN 0-201-13688-0.
- Harbison, Samuel. *Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1992, ISBN 0-13-596396.
- Hoare, C. A. R. *Hints on Programming Language Design*. Stanford University Computer Science Department Technical Report No. CS-73-403, December 1973. Reprinted in SIGACT/SIGPLAN Symposium on Principles of Programming Languages. Association for Computing Machinery, New York, October 1973.
- IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std. 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704 USA; 800-854-7179.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2nd ed. Prentice Hall, Englewood Cliffs, New Jersey, 1988, ISBN 0-13-110362-8.
- Madsen, Ole Lehrmann, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993, ISBN 0-201-62430-3.
- Mitchell, James G., William Maybury, and Richard Sweet. *The Mesa Programming Language, Version 5.0*. Xerox PARC, Palo Alto, California, CSL 79-3, April 1979.
- Stroustrup, Bjarne. *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections January 1994, ISBN 0-201-53992-6.
- Unicode Consortium, The. *The Unicode Standard, Version 6.0.0*. Mountain View, CA, 2011, ISBN 978-1-936213-01-6.

Grammars

THIS chapter describes the context-free grammars used in this specification to define the lexical and syntactic structure of a program.

2.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a language, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

2.2 The Lexical Grammar

A *lexical grammar* for the Java programming language is given in (Chapter 3, *Lexical Structure*). This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Input* (§3.5), that describe how sequences of Unicode characters (§3.1) are translated into a sequence of input elements (§3.5).

These input elements, with white space (§3.6) and comments (§3.7) discarded, form the terminal symbols for the syntactic grammar for the Java programming language and are called *tokens* (§3.5). These tokens are the identifiers (§3.8),

keywords (§3.9), literals (§3.10), separators (§3.11), and operators (§3.12) of the Java programming language.

2.3 The Syntactic Grammar

A *syntactic grammar* for the Java programming language is given in Chapters 4, 6-10, 14, and 15. This grammar has tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions, starting from the goal symbol *CompilationUnit* (§7.3), that describe how sequences of tokens can form syntactically correct programs.

2.4 Grammar Notation

Terminal symbols are shown in `fixed width` font in the productions of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines.

The subscripted suffix "*opt*", which may appear after a terminal or nonterminal, indicates an *optional symbol*. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it.

A very long right-hand side may be continued on a second line by substantially indenting this second line.

When the words "one of" follow the colon in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition.

When an alternative in a lexical production appears to be a token, it represents the sequence of characters that would make up such a token.

The right-hand side of a lexical production may specify that certain expansions are not permitted by using the phrase "but not" and then indicating the expansions to be excluded.

Finally, a few nonterminal symbols are described by a descriptive phrase in roman type in cases where it would be impractical to list all the alternatives.

Lexical Structure

THIS chapter specifies the lexical structure of the Java programming language.

Programs are written in Unicode (§3.1), but lexical translations are provided (§3.2) so that Unicode escapes (§3.3) can be used to include any Unicode character using only ASCII characters. Line terminators are defined (§3.4) to support the different conventions of existing host systems while maintaining consistent line numbers.

The Unicode characters resulting from the lexical translations are reduced to a sequence of input elements (§3.5), which are white space (§3.6), comments (§3.7), and tokens. The tokens are the identifiers (§3.8), keywords (§3.9), literals (§3.10), separators (§3.11), and operators (§3.12) of the syntactic grammar.

3.1 Unicode

Programs are written using the Unicode character set. Information about this character set and its associated character encodings may be found at <http://www.unicode.org/>.

The Java SE platform tracks the Unicode specification as it evolves. The precise version of Unicode used by a given release is specified in the documentation of the class `Character`.

The Unicode standard was originally designed as a fixed-width 16-bit character encoding. It has since been changed to allow for characters whose representation requires more than 16 bits. The range of legal code points is now U+0000 to U+10FFFF, using the hexadecimal *U+n notation*. Characters whose code points are greater than U+FFFF are called supplementary characters. To represent the complete range of characters using only 16-bit units, the Unicode standard defines an encoding called UTF-16. In this encoding, supplementary characters are represented as pairs of 16-bit code units, the first from the high-surrogates range,

(U+D800 to U+DBFF), the second from the low-surrogates range (U+DC00 to U+DFFF). For characters in the range U+0000 to U+FFFF, the values of code points and UTF-16 code units are the same.

The Java programming language represents text in sequences of 16-bit code units, using the UTF-16 encoding. A few APIs, primarily in the Character class, use 32-bit integers to represent code points as individual entities. The Java SE platform provides methods to convert between the two representations.

This specification uses the terms *code point* and *UTF-16 code unit* where the representation is relevant, and the generic term *character* where the representation is irrelevant to the discussion.

Except for comments (§3.7), identifiers, and the contents of character and string literals (§3.10.4, §3.10.5), all input elements (§3.5) in a program are formed only from ASCII characters (or Unicode escapes (§3.3) which result in ASCII characters). ASCII (ANSI X3.4) is the American Standard Code for Information Interchange. The first 128 characters of the Unicode character encoding are the ASCII characters.

3.2 Lexical Translations

A raw Unicode character stream is translated into a sequence of tokens, using the following three lexical translation steps, which are applied in turn:

1. A translation of Unicode escapes (§3.3) in the raw stream of Unicode characters to the corresponding Unicode character. A Unicode escape of the form `\uxxxx`, where `xxxx` is a hexadecimal value, represents the UTF-16 code unit whose encoding is `xxxx`. This translation step allows any program to be expressed using only ASCII characters.
2. A translation of the Unicode stream resulting from step 1 into a stream of input characters and line terminators (§3.4).
3. A translation of the stream of input characters and line terminators resulting from step 2 into a sequence of input elements (§3.5) which, after white space (§3.6) and comments (§3.7) are discarded, comprise the tokens (§3.5) that are the terminal symbols of the syntactic grammar (§2.3).

The longest possible translation is used at each step, even if the result does not ultimately make a correct program while another lexical translation would. Thus the input characters `a--b` are tokenized (§3.5) as `a`, `--`, `b`, which is not part of any

grammatically correct program, even though the tokenization `a, -, -, b` could be part of a grammatically correct program.

3.3 Unicode Escapes

A compiler for the Java programming language ("Java compiler") first recognizes Unicode escapes in its input, translating the ASCII characters `\u` followed by four hexadecimal digits to the UTF-16 code unit (§3.1) of the indicated hexadecimal value, and passing all other characters unchanged. Representing supplementary characters requires two consecutive Unicode escapes. This translation step results in a sequence of Unicode input characters.

UnicodeInputCharacter:

UnicodeEscape

RawInputCharacter

UnicodeEscape:

`\ UnicodeMarker HexDigit HexDigit HexDigit HexDigit`

UnicodeMarker:

`u`

UnicodeMarker `u`

RawInputCharacter:

any Unicode character

HexDigit: one of

`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

The `\`, `u`, and hexadecimal digits here are all ASCII characters.

In addition to the processing implied by the grammar, for each raw input character that is a backslash `\`, input processing must consider how many other `\` characters contiguously precede it, separating it from a non-`\` character or the start of the input stream. If this number is even, then the `\` is eligible to begin a Unicode escape; if the number is odd, then the `\` is not eligible to begin a Unicode escape.

If an eligible `\` is not followed by `u`, then it is treated as a *RawInputCharacter* and remains part of the escaped Unicode stream.

If an eligible `\` is followed by `u`, or more than one `u`, and the last `u` is not followed by four hexadecimal digits, then a compile-time error occurs.

The character produced by a Unicode escape does not participate in further Unicode escapes.

The Java programming language specifies a standard way of transforming a program written in Unicode into ASCII that changes a program into a form that can be processed by ASCII-based tools. The transformation involves converting any Unicode escapes in the source text of the program to ASCII by adding an extra `u` - for example, `\uxxxx` becomes `\uuxxxx` - while simultaneously converting non-ASCII characters in the source text to Unicode escapes containing a single `u` each.

This transformed version is equally acceptable to a Java compiler and represents the exact same program. The exact Unicode source can later be restored from this ASCII form by converting each escape sequence where multiple `u`'s are present to a sequence of Unicode characters with one fewer `u`, while simultaneously converting each escape sequence with a single `u` to the corresponding single Unicode character.

3.4 Line Terminators

A Java compiler next divides the sequence of Unicode input characters into lines by recognizing *line terminators*.

LineTerminator:

- the ASCII LF character, also known as "newline"
- the ASCII CR character, also known as "return"
- the ASCII CR character followed by the ASCII LF character

InputCharacter:

UnicodeInputCharacter but not CR or LF

Lines are terminated by the ASCII characters CR, or LF, or CR LF. The two characters CR immediately followed by LF are counted as one line terminator, not two.

A line terminator specifies the termination of the `//` form of a comment (§3.7).

The result is a sequence of line terminators and input characters, which are the terminal symbols for the third step in the tokenization process.

3.5 Input Elements and Tokens

The input characters and line terminators that result from escape processing (§3.3) and then input line recognition (§3.4) are reduced to a sequence of *input elements*. Those input elements that are not white space (§3.6) or comments (§3.7) are *tokens*. The tokens are the terminal symbols of the syntactic grammar (§2.3).

Input:

*InputElements*_{opt} *Sub*_{opt}

InputElements:

InputElement

InputElements InputElement

InputElement:

WhiteSpace

Comment

Token

Token:

Identifier

Keyword

Literal

Separator

Operator

Sub:

the ASCII SUB character, also known as "control-Z"

White space (§3.6) and comments (§3.7) can serve to separate tokens that, if adjacent, might be tokenized in another manner. For example, the ASCII characters - and = in the input can form the operator token -= (§3.12) only if there is no intervening white space or comment.

As a special concession for compatibility with certain operating systems, the ASCII SUB character (\u001a, or control-Z) is ignored if it is the last character in the escaped input stream.

Consider two tokens *x* and *y* in the resulting input stream. If *x* precedes *y*, then we say that *x* is *to the left of y* and that *y* is *to the right of x*.

3.6 White Space

White space is defined as the ASCII space character, horizontal tab character, form feed character, and line terminator characters (§3.4).

WhiteSpace:

the ASCII SP character, also known as "space"

the ASCII HT character, also known as "horizontal tab"

the ASCII FF character, also known as "form feed"

LineTerminator

3.7 Comments

There are two kinds of comments.

- */* text */*

A *traditional comment*: all the text from the ASCII characters */** to the ASCII characters **/* is ignored (as in C and C++).

- *// text*

An *end-of-line comment*: all the text from the ASCII characters *//* to the end of the line is ignored (as in C++).

Comment:

TraditionalComment
EndOfLineComment

TraditionalComment:

/ CommentTail*

EndOfLineComment:

// CharactersInLine_{opt}

CommentTail:

** CommentTailStar*
NotStar CommentTail

CommentTailStar:

/
** CommentTailStar*
NotStarNotSlash CommentTail

NotStar:

InputCharacter but not ***
LineTerminator

NotStarNotSlash:

InputCharacter but not *** or */*
LineTerminator

CharactersInLine:

InputCharacter
CharactersInLine InputCharacter

These productions imply all of the following properties:

- Comments do not nest.
- */** and **/* have no special meaning in comments that begin with *//*.
- *//* has no special meaning in comments that begin with */** or */***.

3.8 Identifiers

An *identifier* is an unlimited-length sequence of *Java letters* and *Java digits*, the first of which must be a Java letter.

An identifier cannot have the same spelling (Unicode character sequence) as a keyword (§3.9), boolean literal (§3.10.3), or the null literal (§3.10.7).

Identifier:

IdentifierChars but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

IdentifierChars:

JavaLetter

IdentifierChars JavaLetterOrDigit

JavaLetter:

any Unicode character that is a Java letter (see below)

JavaLetterOrDigit:

any Unicode character that is a Java letter-or-digit (see below)

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows programmers to use identifiers in their programs that are written in their native languages.

A "Java letter" is a character for which the method `Character.isJavaIdentifierStart(int)` returns true. A "Java letter-or-digit" is a character for which the method `Character.isJavaIdentifierPart(int)` returns true.

Two identifiers are the same only if they are identical, that is, have the same Unicode character for each letter or digit. Identifiers that have the same external appearance may yet be different.

3.9 Keywords

50 character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers (§3.8).

Keyword: one of

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

3.10 Literals

A *literal* is the source code representation of a value of a primitive type (§4.2), the String type (§4.3.3), or the null type (§4.1).

Literal:

IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

3.10.1 Integer Literals

An *integer literal* may be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8).

IntegerLiteral:

DecimalIntegerLiteral

HexIntegerLiteral

OctalIntegerLiteral

DecimalIntegerLiteral:

DecimalNumeral IntegerTypeSuffix_{opt}

HexIntegerLiteral:

HexNumeral IntegerTypeSuffix_{opt}

OctalIntegerLiteral:

OctalNumeral IntegerTypeSuffix_{opt}

IntegerTypeSuffix: one of

l L

An integer literal is of type `long` if it is suffixed with an ASCII letter `L` or `l` (ell); otherwise it is of type `int` (§4.2.1).

A decimal numeral is either the single ASCII character `0`, representing the integer zero, or consists of an ASCII digit from `1` to `9`, optionally followed by one or more ASCII digits from `0` to `9`, representing a positive integer.

DecimalNumeral:

0

NonZeroDigit Digits_{opt}

Digits:

Digit

Digits Digit

Digit:

0

NonZeroDigit

NonZeroDigit: one of

1 2 3 4 5 6 7 8 9

A hexadecimal numeral consists of the leading ASCII characters `0x` or `0X` followed by one or more ASCII hexadecimal digits and can represent a positive, zero, or negative integer. Hexadecimal digits with values `10` through `15` are represented by

the ASCII letters a through f or A through F, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

HexNumeral:

0 x *HexDigits*

0 *HexDigits*

HexDigits:

HexDigit

HexDigits HexDigit

HexDigit: one of

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

An octal numeral consists of an ASCII digit 0 followed by one or more of the ASCII digits 0 through 7 and can represent a positive, zero, or negative integer.

OctalNumeral:

0 *OctalDigits*

OctalDigits:

OctalDigit

OctalDigits OctalDigit

OctalDigit: one of

0 1 2 3 4 5 6 7

The largest decimal literal of type `int` is 2147483648 (2^{31}). All decimal literals from 0 to 2147483647 may appear anywhere an `int` literal may appear, but the literal 2147483648 may appear only as the operand of the unary negation operator `-`.

The largest positive hexadecimal and octal literals of type `int` are `0x7fffffff` and `017777777777`, respectively, which equal 2147483647 ($2^{31}-1$).

The most negative hexadecimal and octal literals of type `int` are `0x80000000` and `020000000000`, respectively, each of which represents the decimal value -2147483648 (-2^{31}). The hexadecimal and octal literals `0xffffffff` and `037777777777`, respectively, represent the decimal value -1.

It is a compile-time error if a decimal literal of type `int` is larger than 2147483648 (2^{31}), or if the literal 2147483648 appears anywhere other than as the operand of the unary `-` operator, or if a hexadecimal or octal `int` literal does not fit in 32 bits.

The largest decimal literal of type `long` is `9223372036854775808L` (2^{63}). All decimal literals from `0L` to `9223372036854775807L` may appear anywhere a `long` literal may appear, but the literal `9223372036854775808L` may appear only as the operand of the unary negation operator `-`.

The largest positive hexadecimal and octal literals of type `long` are `0x7fffffffffffffffffL` and `07777777777777777777L`, respectively, which equal `9223372036854775807L` ($2^{63}-1$).

The most negative hexadecimal and octal literals of type `long` are `0x8000000000000000L` and `0100000000000000000000L`, respectively. Each has the decimal value `-9223372036854775808L` (-2^{63}). The hexadecimal and octal literals `0xfffffffffffffffffL` and `01777777777777777777L`, respectively, represent the decimal value `-1L`.

It is a compile-time error if a decimal literal of type `long` is larger than `9223372036854775808L` (2^{63}), or if the literal `9223372036854775808L` appears anywhere other than as the operand of the unary `-` operator, or if a hexadecimal or octal `long` literal does not fit in 64 bits.

3.10.2 Floating-Point Literals

A *floating-point literal* has the following parts: a whole-number part, a decimal or hexadecimal point (represented by an ASCII period character), a fractional part, an exponent, and a type suffix.

A floating point number may be written either as a decimal value or as a hexadecimal value. For decimal literals, the exponent, if present, is indicated by the ASCII letter `e` or `E` followed by an optionally signed integer. For hexadecimal literals, the exponent is always required and is indicated by the ASCII letter `p` or `P` followed by an optionally signed integer.

For decimal floating-point literals, at least one digit, in either the whole number or the fraction part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional. For hexadecimal floating-point literals, at least one digit is required in either the whole number or fraction part, the exponent is mandatory, and the float type suffix is optional.

A floating-point literal is of type `float` if it is suffixed with an ASCII letter `F` or `f`; otherwise its type is `double` and it can optionally be suffixed with an ASCII letter `D` or `d`.

FloatingPointLiteral:

DecimalFloatingPointLiteral

HexadecimalFloatingPointLiteral

DecimalFloatingPointLiteral:

Digits . Digits_{opt} ExponentPart_{opt} FloatTypeSuffix_{opt}

. Digits ExponentPart_{opt} FloatTypeSuffix_{opt}

Digits ExponentPart FloatTypeSuffix_{opt}

Digits ExponentPart_{opt} FloatTypeSuffix

ExponentPart:

ExponentIndicator SignedInteger

ExponentIndicator: one of

e E

SignedInteger:

Sign_{opt} Digits

Sign: one of

+ -

FloatTypeSuffix: one of

f F d D

HexadecimalFloatingPointLiteral:

HexSignificand BinaryExponent FloatTypeSuffix_{opt}

HexSignificand:

HexNumeral

HexNumeral .

0x HexDigits_{opt} . HexDigits

0X HexDigits_{opt} . HexDigits

BinaryExponent:

BinaryExponentIndicator SignedInteger

BinaryExponentIndicator: one of

p P

The elements of the types `float` and `double` are those values that can be represented using the IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point formats, respectively.

The largest positive finite literal of type `float` is `3.4028235e38f`. The smallest positive finite nonzero literal of type `float` is `1.40e-45f`.

The largest positive finite literal of type `double` is `1.7976931348623157e308`. The smallest positive finite nonzero literal of type `double` is `4.9e-324`.

It is a compile-time error if a nonzero floating-point literal is too large, so that on rounded conversion to its internal representation, it becomes an IEEE 754 infinity.

It is a compile-time error if a nonzero floating-point literal is too small, so that, on rounded conversion to its internal representation, it becomes a zero.

Predefined constants representing Not-a-Number values are defined in the classes `Float` and `Double` as `Float.NaN` and `Double.NaN`.

3.10.3 Boolean Literals

The `boolean` type has two values, represented by the literals `true` and `false`, formed from ASCII letters.

A *boolean literal* is always of type `boolean`.

BooleanLiteral: one of
`true false`

3.10.4 Character Literals

A *character literal* is expressed as a character or an escape sequence (§3.10.6), enclosed in ASCII single quotes. (The single-quote, or apostrophe, character is `\u0027`.)

Character literals can only represent UTF-16 code units (§3.1), i.e., they are limited to values from `\u0000` to `\uffff`. Supplementary characters must be represented either as a surrogate pair within a char sequence, or as an integer, depending on the API they are used with.

A character literal is always of type `char`.

CharacterLiteral:

' *SingleCharacter* '
' *EscapeSequence* '

SingleCharacter:

InputCharacter but not ' or \

As specified in §3.4, the characters CR and LF are never an *InputCharacter*; they are recognized as constituting a *LineTerminator*.

It is a compile-time error for the character following the *SingleCharacter* or *EscapeSequence* to be other than a '.

It is a compile-time error for a line terminator to appear after the opening ' and before the closing '.

In C and C++, a character literal may contain representations of more than one character, but the value of such a character literal is implementation-defined. In the Java programming language, a character literal always represents exactly one character.

3.10.5 String Literals

A *string literal* consists of zero or more characters enclosed in double quotes. Characters may be represented by escape sequences (§3.10.6) - one escape sequence for characters in the range U+0000 to U+FFFF, two escape sequences for the UTF-16 surrogate code units of characters in the range U+010000 to U+10FFFF.

A string literal is always of type `String` (§4.3.3).

A string literal always refers to the same instance (§4.3.1) of class `String`.

StringLiteral:

" *StringCharacters_{opt}* "

StringCharacters:

StringCharacter
StringCharacters StringCharacter

StringCharacter:

InputCharacter but not " or \
EscapeSequence

As specified in §3.4, neither of the characters CR and LF is ever considered to be an *InputCharacter*; each is recognized as constituting a *LineTerminator*.

It is a compile-time error for a line terminator to appear after the opening " and before the closing matching ". A long string literal can always be broken up into shorter pieces and written as a (possibly parenthesized) expression using the string concatenation operator + (§15.18.1).

Each string literal is a reference (§4.3) to an instance (§4.3.1, §12.5) of class `String` (§4.3.3). `String` objects have a constant value. String literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method `String.intern`.

3.10.6 Escape Sequences for Character and String Literals

The character and string *escape sequences* allow for the representation of some nongraphic characters as well as the single quote, double quote, and backslash characters in character literals (§3.10.4) and string literals (§3.10.5).

EscapeSequence:

```
\ b  /* \u0008: backspace BS */
\ t  /* \u0009: horizontal tab HT */
\ n  /* \u000a: linefeed LF */
\ f  /* \u000c: form feed FF */
\ r  /* \u000d: carriage return CR */
\ "  /* \u0022: double quote " */
\ '  /* \u0027: single quote ' */
\ \      /* \u005c: backslash \ */
OctalEscape      /* \u0000 to \u00ff: from octal value */
```

OctalEscape:

```
\ OctalDigit
\ OctalDigit OctalDigit
\ ZeroToThree OctalDigit OctalDigit
```

OctalDigit: one of

```
0 1 2 3 4 5 6 7
```

ZeroToThree: one of

```
0 1 2 3
```

It is a compile-time error if the character following a backslash in an escape is not an ASCII b, t, n, f, r, ", ', \, 0, 1, 2, 3, 4, 5, 6, or 7. The Unicode escape \u is processed earlier (§3.3). (Octal escapes are provided for compatibility with C, but can express only Unicode values \u0000 through \u00FF, so Unicode escapes are usually preferred.)

3.10.7 The Null Literal

The null type has one value, the null reference, represented by the literal `null`, which is formed from ASCII characters. A *null literal* is always of the null type.

NullLiteral:
`null`

3.11 Separators

Nine ASCII characters are the *separators* (punctuators).

Separator: one of
`( ) { } [ ] ; , .`

3.12 Operators

37 tokens are the *operators*, formed from ASCII characters.

Operator: one of
`= > < ! ~ ? :`
`== <= >= != && || ++ --`
`+ - * / & | ^ % << >> >>>`
`+= -= *= /= &= |= ^= %= <<= >>= >>>=`

Types, Values, and Variables

THE Java programming language is a *strongly typed* language, which means that every variable and every expression has a type that is known at compile time. Types limit the values that a variable (§4.12) can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong typing helps detect errors at compile time.

The types of the Java programming language are divided into two categories: primitive types and reference types. The primitive types (§4.2) are the `boolean` type and the numeric types. The numeric types are the integral types `byte`, `short`, `int`, `long`, and `char`, and the floating-point types `float` and `double`. The reference types (§4.3) are class types, interface types, and array types. There is also a special null type. An object (§4.3.1) is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to objects. All objects, including arrays, support the methods of class `Object` (§4.3.2). String literals are represented by `String` objects (§4.3.3).

Types exist at compile-time. Some types correspond to classes and interfaces, which exist at run-time. The correspondence between types and classes or interfaces is incomplete for two reasons:

1. At run-time, classes and interfaces are loaded by the Java virtual machine using class loaders. Each class loader defines its own set of classes and interfaces. As a result, it is possible for two loaders to load an identical class or interface definition but produce distinct classes or interfaces at run-time.

Consequently, code that compiled correctly may fail at link time if the class loaders that load it are inconsistent. See the paper *Dynamic Class Loading in the Java™ Virtual Machine*, by Sheng Liang and Gilad Bracha, in *Proceedings of OOPSLA '98*, published as *ACM SIGPLAN Notices*, Volume 33, Number 10, October 1998, pages 36-44, and *The Java Virtual Machine Specification* for more details.

2. Type variables (§4.4) and type arguments (§4.5.1) are not reified at run-time. As a result, the same class or interface at run-time represents different parameterized types (§4.5) from compile-time. Specifically, all compile-time invocations of a given generic type declaration (§8.1.2, §9.1.2) share a single run-time representation.

4.1 The Kinds of Types and Values

There are two kinds of types in the Java programming language: primitive types (§4.2) and reference types (§4.3). There are, correspondingly, two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values (§4.2) and reference values (§4.3).

Type:

PrimitiveType

ReferenceType

There is also a special *null type*, the type of the expression `null`, which has no name.

Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type.

The null reference is the only possible value of an expression of null type.

The null reference can always be cast to any reference type.

4.2 Primitive Types and Values

A primitive type is predefined by the Java programming language and named by its reserved keyword (§3.9):

PrimitiveType:

NumericType

`boolean`

NumericType:

IntegralType

FloatingPointType

IntegralType: one of

`byte short int long char`

FloatingPointType: one of

`float double`

Primitive values do not share state with other primitive values.

A variable whose type is a primitive type always holds a primitive value of that same type.

The value of a variable of primitive type can be changed only by assignment operations on that variable (including increment (§15.14.2, §15.15.1) and decrement (§15.14.3, §15.15.2) operators).

The *numeric types* are the integral types and the floating-point types.

The *integral types* are `byte`, `short`, `int`, and `long`, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and `char`, whose values are 16-bit unsigned integers representing UTF-16 code units (§3.1).

The *floating-point types* are `float`, whose values include the 32-bit IEEE 754 floating-point numbers, and `double`, whose values include the 64-bit IEEE 754 floating-point numbers.

The `boolean` type has exactly two values: `true` and `false`.

4.2.1 Integral Types and Values

The values of the integral types are integers in the following ranges:

- For `byte`, from -128 to 127, inclusive
- For `short`, from -32768 to 32767, inclusive
- For `int`, from -2147483648 to 2147483647, inclusive
- For `long`, from -9223372036854775808 to 9223372036854775807, inclusive

- For char, from '\u0000' to '\uffff' inclusive, that is, from 0 to 65535

4.2.2 Integer Operations

The Java programming language provides a number of operators that act on integral values:

- The comparison operators, which result in a value of type `boolean`:
 - ♦ The numerical comparison operators `<`, `<=`, `>`, and `>=` (§15.20.1)
 - ♦ The numerical equality operators `==` and `!=` (§15.21.1)
- The numerical operators, which result in a value of type `int` or `long`:
 - ♦ The unary plus and minus operators `+` and `-` (§15.15.3, §15.15.4)
 - ♦ The multiplicative operators `*`, `/`, and `%` (§15.17)
 - ♦ The additive operators `+` and `-` (§15.18)
 - ♦ The increment operator `++`, both prefix (§15.15.1) and postfix (§15.14.2)
 - ♦ The decrement operator `--`, both prefix (§15.15.2) and postfix (§15.14.3)
 - ♦ The signed and unsigned shift operators `<<`, `>>`, and `>>>` (§15.19)
 - ♦ The bitwise complement operator `~` (§15.15.5)
 - ♦ The integer bitwise operators `&`, `|`, and `^` (§15.22.1)
- The conditional operator `? :` (§15.25)
- The cast operator, which can convert from an integral value to a value of any specified numeric type (§5.5, §15.16)
- The string concatenation operator `+` (§15.18.1), which, when given a `String` operand and an integral operand, will convert the integral operand to a `String` representing its value in decimal form, and then produce a newly created `String` that is the concatenation of the two strings

Other useful constructors, methods, and constants are predefined in the classes `Byte`, `Short`, `Integer`, `Long`, and `Character`.

If an integer operator other than a shift operator has at least one operand of type `long`, then the operation is carried out using 64-bit precision, and the result of the numerical operator is of type `long`. If the other operand is not `long`, it is first widened (§5.1.5) to type `long` by numeric promotion (§5.6).

Otherwise, the operation is carried out using 32-bit precision, and the result of the numerical operator is of type `int`. If either operand is not an `int`, it is first widened to type `int` by numeric promotion.

The built-in integer operators do not indicate overflow or underflow in any way.

Integer operators can throw a `NullPointerException` if unboxing conversion (§5.1.8) of a null reference is required.

Other than that, the only integer operators that can throw an exception (Chapter 11, *Exceptions*) are the integer divide operator `/` (§15.17.2) and the integer remainder operator `%` (§15.17.3), which throw an `ArithmeticException` if the right-hand operand is zero, and the increment and decrement operators `++` (§15.15.1, §15.15.2) and `--` (§15.14.3, §15.14.2), which can throw an `OutOfMemoryError` if boxing conversion (§5.1.7) is required and there is not sufficient memory available to perform the conversion.

Any value of any integral type may be cast to or from any numeric type. There are no casts between integral types and the type `boolean`.

4.2.3 Floating-Point Types, Formats, and Values

The floating-point types are `float` and `double`, which are conceptually associated with the single-precision 32-bit and double-precision 64-bit format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative numbers that consist of a sign and magnitude, but also positive and negative zeros, positive and negative *infinities*, and special *Not-a-Number* values (hereafter abbreviated NaN). A NaN value is used to represent the result of certain invalid operations such as dividing zero by zero. NaN constants of both `float` and `double` type are predefined as `Float.NaN` and `Double.NaN`.

Every implementation of the Java programming language is required to support two standard sets of floating-point values, called the *float value set* and the *double value set*. In addition, an implementation of the Java programming language may support either or both of two extended-exponent floating-point value sets, called the *float-extended-exponent value set* and the *double-extended-exponent value set*. These extended-exponent value sets may, under certain circumstances, be used instead of the standard value sets to represent the values of expressions of type `float` or `double` (§5.1.13, §15.4).

The finite nonzero values of any floating-point value set can all be expressed in the form $s \cdot m \cdot 2^{(e - N + 1)}$, where s is +1 or -1, m is a positive integer less than 2^N , and e is an integer between $E_{min} = -(2^{K-1}-2)$ and $E_{max} = 2^{K-1}-1$, inclusive, and where N and K are parameters that depend on the value set. Some values can be represented in this form in more than one way; for example, supposing that a value v in a value set might be represented in this form using certain values for s , m , and e , then if it happened that m were even and e were less than 2^{K-1} , one could halve m and increase e by 1 to produce a second representation for the same value v . A representation in this form is called *normalized* if $m \geq 2^{(N-1)}$; otherwise the representation is said to be *denormalized*. If a value in a value set cannot be represented in such a way that $m \geq 2^{(N-1)}$, then the value is said to be a *denormalized value*, because it has no normalized representation.

The constraints on the parameters N and K (and on the derived parameters E_{min} and E_{max}) for the two required and two optional floating-point value sets are summarized in Table 4.1.

Table 4.1. Floating-point value set parameters

Parameter	float	float-extended-exponent	double	double-extended-exponent
N	24	24	53	53
K	8	≥ 11	≥ 11	15
E_{max}	+127	$\geq +1023$	+1023	$\geq +16383$
E_{min}	-126	≤ -1022	-1022	≤ -16382

Where one or both extended-exponent value sets are supported by an implementation, then for each supported extended-exponent value set there is a specific implementation-dependent constant K , whose value is constrained by Table 4.1; this value K in turn dictates the values for E_{min} and E_{max} .

Each of the four value sets includes not only the finite nonzero values that are ascribed to it above, but also NaN values and the four values positive zero, negative zero, positive infinity, and negative infinity.

Note that the constraints in Table 4.1 are designed so that every element of the float value set is necessarily also an element of the float-extended-exponent value set, the double value set, and the double-extended-exponent value set. Likewise, each element of the double value set is necessarily also an element of the double-extended-exponent value set. Each extended-exponent value set has a larger range

of exponent values than the corresponding standard value set, but does not have more precision.

The elements of the float value set are exactly the values that can be represented using the single floating-point format defined in the IEEE 754 standard. The elements of the double value set are exactly the values that can be represented using the double floating-point format defined in the IEEE 754 standard. Note, however, that the elements of the float-extended-exponent and double-extended-exponent value sets defined here do *not* correspond to the values that can be represented using IEEE 754 single extended and double extended formats, respectively.

The float, float-extended-exponent, double, and double-extended-exponent value sets are not types. It is always correct for an implementation of the Java programming language to use an element of the float value set to represent a value of type `float`; however, it may be permissible in certain regions of code for an implementation to use an element of the float-extended-exponent value set instead. Similarly, it is always correct for an implementation to use an element of the double value set to represent a value of type `double`; however, it may be permissible in certain regions of code for an implementation to use an element of the double-extended-exponent value set instead.

Except for NaN, floating-point values are *ordered*; arranged from smallest to largest, they are negative infinity, negative finite nonzero values, positive and negative zero, positive finite nonzero values, and positive infinity.

IEEE 754 allows multiple distinct NaN values for each of its single and double floating-point formats. While each hardware architecture returns a particular bit pattern for NaN when a new NaN is generated, a programmer can also create NaNs with different bit patterns to encode, for example, retrospective diagnostic information.

For the most part, the Java SE platform treats NaN values of a given type as though collapsed into a single canonical value (and hence this specification normally refers to an arbitrary NaN as though to a canonical value). However, version 1.3 of the Java SE platform introduced methods enabling the programmer to distinguish between NaN values: the `Float.floatToRawIntBits` and `Double.doubleToRawLongBits` methods. The interested reader is referred to the specifications for the `Float` and `Double` classes for more information.

Positive zero and negative zero compare equal; thus the result of the expression `0.0== -0.0` is `true` and the result of `0.0> -0.0` is `false`. But other operations can distinguish positive and negative zero; for example, `1.0/0.0` has the value positive infinity, while the value of `1.0/-0.0` is negative infinity.

NaN is *unordered*, so the numerical comparison operators `<`, `<=`, `>`, and `>=` return `false` if either or both operands are NaN (§15.20.1). The equality operator `==` returns `false` if either operand is NaN, and the inequality operator `!=` returns `true` if either operand is NaN (§15.21.1). In particular, `x!=x` is `true` if and only if `x` is NaN, and `(x<y) == !(x>=y)` will be `false` if `x` or `y` is NaN.

Any value of a floating-point type may be cast to or from any numeric type. There are no casts between floating-point types and the type `boolean`.

4.2.4 Floating-Point Operations

The Java programming language provides a number of operators that act on floating-point values:

- The comparison operators, which result in a value of type `boolean`:
 - ♦ The numerical comparison operators `<`, `<=`, `>`, and `>=` (§15.20.1)
 - ♦ The numerical equality operators `==` and `!=` (§15.21.1)
- The numerical operators, which result in a value of type `float` or `double`:
 - ♦ The unary plus and minus operators `+` and `-` (§15.15.3, §15.15.4)
 - ♦ The multiplicative operators `*`, `/`, and `%` (§15.17)
 - ♦ The additive operators `+` and `-` (§15.18.2)
 - ♦ The increment operator `++`, both prefix (§15.15.1) and postfix (§15.14.2)
 - ♦ The decrement operator `--`, both prefix (§15.15.2) and postfix (§15.14.3)
- The conditional operator `? :` (§15.25)
- The cast operator, which can convert from a floating-point value to a value of any specified numeric type (§5.5, §15.16)
- The string concatenation operator `+` (§15.18.1), which, when given a `String` operand and a floating-point operand, will convert the floating-point operand to a `String` representing its value in decimal form (without information loss), and then produce a newly created `String` by concatenating the two strings

Other useful constructors, methods, and constants are predefined in the classes `Float`, `Double`, and `Math`.

If at least one of the operands to a binary operator is of floating-point type, then the operation is a floating-point operation, even if the other is integral.

If at least one of the operands to a numerical operator is of type `double`, then the operation is carried out using 64-bit floating-point arithmetic, and the result of the numerical operator is a value of type `double`. (If the other operand is not a `double`, it is first widened to type `double` by numeric promotion (§5.6).) Otherwise, the operation is carried out using 32-bit floating-point arithmetic, and the result of the numerical operator is a value of type `float`. If the other operand is not a `float`, it is first widened to type `float` by numeric promotion.

Operators on floating-point numbers behave as specified by IEEE 754 (with the exception of the remainder operator (§15.17.3)). In particular, the Java programming language requires support of IEEE 754 *denormalized* floating-point numbers and *gradual underflow*, which make it easier to prove desirable properties of particular numerical algorithms. Floating-point operations do not "flush to zero" if the calculated result is a denormalized number.

The Java programming language requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision. *Inexact* results must be rounded to the representable value nearest to the infinitely precise result; if the two nearest representable values are equally near, the one with its least significant bit zero is chosen. This is the IEEE 754 standard's default rounding mode known as *round to nearest*.

The language uses *round toward zero* when converting a floating value to an integer (§5.1.3), which acts, in this case, as though the number were truncated, discarding the mantissa bits. Rounding toward zero chooses as its result the format's value closest to and no greater in magnitude than the infinitely precise result.

Floating-point operators can throw a `NullPointerException` if unboxing conversion (§5.1.8) of a null reference is required. Other than that, the only floating-point operators that can throw an exception (Chapter 11, *Exceptions*) are the increment and decrement operators `++` (§15.15.1, §15.15.2) and `--` (§15.14.3, §15.14.2), which can throw an `OutOfMemoryError` if boxing conversion (§5.1.7) is required and there is not sufficient memory available to perform the conversion.

An operation that overflows produces a signed infinity, an operation that underflows produces a denormalized value or a signed zero, and an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result. As has already been described, NaN is unordered, so a numeric comparison operation involving one or two NaNs returns `false` and any `!=` comparison involving NaN returns `true`, including `x!=x` when `x` is NaN.

4.2.5 The boolean Type and boolean Values

The `boolean` type represents a logical quantity with two possible values, indicated by the literals `true` and `false` (§3.10.3). The boolean operators are:

- The relational operators `==` and `!=` (§15.21.2)
- The logical-complement operator `!` (§15.15.6)
- The logical operators `&`, `^`, and `|` (§15.22.2)
- The conditional-and and conditional-or operators `&&` (§15.23) and `||` (§15.24)
- The conditional operator `? :` (§15.25)
- The string concatenation operator `+` (§15.18.1), which, when given a `String` operand and a `boolean` operand, will convert the `boolean` operand to a `String` (either `"true"` or `"false"`), and then produce a newly created `String` that is the concatenation of the two strings

Boolean expressions determine the control flow in several kinds of statements:

- The `if` statement (§14.9)
- The `while` statement (§14.12)
- The `do` statement (§14.13)
- The `for` statement (§14.14)

A `boolean` expression also determines which subexpression is evaluated in the conditional `? :` operator (§15.25).

Only `boolean` and `Boolean` expressions can be used in control flow statements and as the first operand of the conditional operator `? :`.

An integer `x` can be converted to a `boolean`, following the C language convention that any nonzero value is `true`, by the expression `x!=0`.

An object reference `obj` can be converted to a `boolean`, following the C language convention that any reference other than `null` is `true`, by the expression `obj!=null`.

A cast of a `boolean` value to type `boolean` or `Boolean` is allowed (§5.1.1); no other casts on type `boolean` are allowed.

A `boolean` can be converted to a `String` by string conversion (§5.4).

4.3 Reference Types and Values

There are four kinds of *reference types*: class types (Chapter 8, *Classes*), interface types (Chapter 9, *Interfaces*), type variables (§4.4), and array types (Chapter 10, *Arrays*).

ReferenceType:

ClassOrInterfaceType

TypeVariable

ArrayType

ClassOrInterfaceType:

ClassType

InterfaceType

ClassType:

TypeDeclSpecifier TypeArguments_{opt}

InterfaceType:

TypeDeclSpecifier TypeArguments_{opt}

TypeDeclSpecifier:

Identifier

ClassOrInterfaceType . Identifier

TypeName:

Identifier

TypeName . Identifier

TypeVariable:

Identifier

ArrayType:

Type []

A class or interface type consists of a *type declaration specifier*, optionally followed by type arguments (§4.5.1). If type arguments appear anywhere in a class or interface type, it is a parameterized type (§4.5).

A type declaration specifier may be either a type name (§6.5.5), or a class or interface type followed by "." and an identifier. In the latter case, the specifier has

the form *T.id*, where *id* must be the simple name of an accessible (§6.6) member type (§8.5, §9.5) of *T*, or a compile-time error occurs. The specifier denotes that member type.

4.3.1 Objects

An *object* is a *class instance* or an *array*.

The reference values (often just *references*) are pointers to these objects, and a special null reference, which refers to no object.

A class instance is explicitly created by a class instance creation expression (§15.9). An array is explicitly created by an array creation expression (§15.10).

A new class instance is implicitly created when the string concatenation operator + (§15.18.1) is used in a non-constant (§15.28) expression, resulting in a new object of type `String` (§4.3.3).

A new array object is implicitly created when an array initializer expression (§10.6) is evaluated; this can occur when a class or interface is initialized (§12.4), when a new instance of a class is created (§15.9), or when a local variable declaration statement is executed (§14.4).

New objects of the types `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double` may be implicitly created by boxing conversion (§5.1.7).

The operators on references to objects are:

- Field access, using either a qualified name (§6.6) or a field access expression (§15.11)
- Method invocation (§15.12)
- The cast operator (§5.5, §15.16)
- The string concatenation operator + (§15.18.1), which, when given a `String` operand and a reference, will convert the reference to a `String` by invoking the `toString` method of the referenced object (using "null" if either the reference or the result of `toString` is a null reference), and then will produce a newly created `String` that is the concatenation of the two strings
- The `instanceof` operator (§15.20.2)
- The reference equality operators `==` and `!=` (§15.21.3)
- The conditional operator `? :` (§15.25).

There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable.

Each object has an associated lock (§17.1), which is used by synchronized methods (§8.4.3) and the synchronized statement (§14.19) to provide control over concurrent access to state by multiple threads (Chapter 17, *Threads and Locks*).

4.3.2 The Class Object

The class `Object` is a superclass (§8.1) of all other classes.

All class and array types inherit the methods of class `Object`, which are summarized as follows:

- The method `clone` is used to make a duplicate of an object.
- The method `equals` defines a notion of object equality, which is based on value, not reference, comparison.
- The method `finalize` is run just before an object is destroyed (§12.6).
- The method `getClass` returns the `Class` object that represents the class of the object for reflection purposes. A `Class` object exists for each reference type.

The type of a method invocation expression of `getClass` is `Class<? extends T>` where `T` is the class or interface searched (§15.12.1) for `getClass`.

A class method that is declared synchronized (§8.4.3.6) synchronizes on the lock associated with the `Class` object of the class.

- The method `hashCode` is very useful, together with the method `equals`, in hashtables such as `java.util.HashMap`.
- The methods `wait`, `notify`, and `notifyAll` are used in concurrent programming using threads (§17.2).
- The method `toString` returns a `String` representation of the object.

A variable of type `Object` can hold a reference to the null reference or to any object, whether it is an instance of a class or an array (Chapter 10, *Arrays*).

4.3.3 The Class String

Instances of class `String` represent sequences of Unicode code points.

A `String` object has a constant (unchanging) value.

String literals (§3.10.5) are references to instances of class `String`.

The string concatenation operator `+` (§15.18.1) implicitly creates a new `String` object when the result is not a compile-time constant expression (§15.28).

4.3.4 When Reference Types Are the Same

Two reference types are the *same compile-time type* if they have the same binary name (§13.1) and their type arguments, if any, are the same, applying this definition recursively.

When two reference types are the same, they are sometimes said to be the *same class* or the *same interface*.

At run time, several reference types with the same binary name may be loaded simultaneously by different class loaders. These types may or may not represent the same type declaration. Even if two such types do represent the same type declaration, they are considered distinct.

Two reference types are the *same run-time type* if:

- They are both class or both interface types, are defined by the same class loader, and have the same binary name (§13.1), in which case they are sometimes said to be the *same run-time class* or the *same run-time interface*.
- They are both array types, and their component types are the same run-time type (Chapter 10, *Arrays*).

4.4 Type Variables

A *type variable* is an unqualified identifier.

A type variable is known as a *type parameter* when it is introduced by a generic class declaration (§8.1.2), generic interface declaration (§9.1.2), generic method declaration (§8.4.4), or generic constructor declaration (§8.8.4).

TypeParameter:

TypeVariable TypeBound_{opt}

TypeBound:

extends TypeVariable

extends ClassOrInterfaceType AdditionalBoundList_{opt}

AdditionalBoundList:

AdditionalBound AdditionalBoundList

AdditionalBound

AdditionalBound:

& InterfaceType

A type variable has an optional bound, $T \& I_1 \& \dots \& I_n$. The bound consists of either a type variable, or a class or interface type T possibly followed by further interface types $I_1, \dots, I_n$. If no bound is given for a type variable, `Object` is assumed.

It is a compile-time error if any of the types $I_1 \dots I_n$ is a class type or type variable.

The erasures (§4.6) of all constituent types of a bound must be pairwise different, or a compile-time error occurs.

The order of types in a bound is only significant in that the erasure of a type variable is determined by the first type in its bound, and that a class type or type variable may only appear in the first position.

A type variable may not at the same time be a subtype of two interface types which are different parameterizations of the same generic interface.

The members of a type variable X with bound $T \& I_1 \& \dots \& I_n$ are the members of the intersection type (§4.9) $T \& I_1 \& \dots \& I_n$ appearing at the point where the type variable is declared.

4.5 Parameterized Types

A generic class or interface declaration C (§8.1.2, §9.1.2) with one or more type parameters $A_1, \dots, A_n$ which have corresponding bounds $B_1, \dots, B_n$ defines a set of parameterized types, one for each possible invocation of the type parameter section.

A parameterized type is written as a *ClassType* or *InterfaceType* that contains at least one type declaration specifier immediately followed by a type argument list

$\langle T_1, \dots, T_n \rangle$. The type argument list denotes a particular invocation of the type parameters of the generic type indicated by the type declaration specifier.

Given a type declaration specifier immediately followed by a type argument list, let C be the final *Identifier* in the specifier.

It is a compile-time error if C is not the name of a generic class or interface, or if the number of type arguments in the type argument list differs from the number of type parameters of C .

Let $P = C\langle T_1, \dots, T_n \rangle$ be a parameterized type. It must be the case that, after P is subjected to capture conversion (§5.1.10) resulting in the type $C\langle X_1, \dots, X_n \rangle$, for each type argument X_i ($1 \leq i \leq n$), $X_i <: B_i[A_1 := X_1, \dots, A_n := X_n]$ (§4.10), or a compile-time error occurs.

In this specification, whenever we speak of a class or interface type, we include the generic version as well, unless explicitly excluded.

Two parameterized types are provably distinct if either of the following conditions hold:

- They are invocations of distinct generic type declarations.
- Any of their type arguments are provably distinct.

4.5.1 Type Arguments and Wildcards

Type arguments may be either reference types or wildcards. Wildcards are useful in situations where only partial knowledge about the type parameter is required.

TypeArguments:

< TypeArgumentList >

TypeArgumentList:

TypeArgument

TypeArgumentList , TypeArgument

TypeArgument:

ReferenceType

Wildcard

Wildcard:

? WildcardBounds_{opt}

WildcardBounds:

extends ReferenceType

super ReferenceType

Wildcards may be given explicit bounds, just like regular type variable declarations. An upper bound is signified by the syntax:

? extends B

where *B* is the bound.

Unlike ordinary type variables declared in a method signature, no type inference is required when using a wildcard. Consequently, it is permissible to declare lower bounds on a wildcard, using the syntax:

? super B

where *B* is a lower bound.

Two type arguments are provably distinct if one of the following is true:

- Neither argument is a type variable or wildcard, and the two arguments are not the same type.
- One type argument is a type variable or wildcard, with an upper bound (from capture conversion, if necessary) of *S*; and the other type argument *T* is not a type variable or wildcard; and neither $|S| <: |T|$ nor $|T| <: |S|$.
- Each type argument is a type variable or wildcard, with upper bounds (from capture conversion, if necessary) of *S* and *T*; and neither $|S| <: |T|$ nor $|T| <: |S|$.

A type argument T_1 is said to contain another type argument T_2 , written $T_2 \leq T_1$, if the set of types denoted by T_2 is provably a subset of the set of types denoted by T_1 under the reflexive and transitive closure of the following rules (where $<:$ denotes subtyping (§4.10)):

- $? \text{ extends } T \leq ? \text{ extends } S \text{ if } T <: S$
- $? \text{ super } T \leq ? \text{ super } S \text{ if } S <: T$
- $T \leq T$
- $T \leq ? \text{ extends } T$
- $T \leq ? \text{ super } T$

4.5.2 Members and Constructors of Parameterized Types

Let C be a generic class or interface declaration with type parameters $A_1, \dots, A_n$, and let $C<T_1, \dots, T_n>$ be an invocation of C , where, for $1 \leq i \leq n$, T_i are types (rather than wildcards). Then:

- Let m be a member or constructor declaration (§8.2, §8.8.6) in C , whose type as declared is T . Then the type of m in $C<T_1, \dots, T_n>$, is $T[A_1 := T_1, \dots, A_n := T_n]$.
- Let m be a member or constructor declaration in D , where D is a class extended by C or an interface implemented by C . Let $D<U_1, \dots, U_k>$ be the supertype of $C<T_1, \dots, T_n>$ that corresponds to D . Then the type of m in $C<T_1, \dots, T_n>$ is the type of m in $D<U_1, \dots, U_k>$.

If any of the type arguments in the invocation of C are wildcards, then:

- The types of the fields, methods, and constructors in $C<T_1, \dots, T_n>$ are undefined.
- Let D be a (possibly generic) class or interface declaration in C . Then the type of D in $C<T_1, \dots, T_n>$ is D where, if D is generic, all type arguments are unbounded wildcards.

4.6 Type Erasure

Type erasure is a mapping from types (possibly including parameterized types and type variables) to types (that are never parameterized types or type variables). We write $|T|$ for the erasure of type T . The erasure mapping is defined as follows.

- The erasure of a parameterized type (§4.5) $G<T_1, \dots, T_n>$ is $|G|$.

- The erasure of a nested type $T.C$ is $|T|.C$.
- The erasure of an array type $T[]$ is $|T|[]$.
- The erasure of a type variable (§4.4) is the erasure of its leftmost bound.
- The erasure of every other type is the type itself.

Type erasure also maps the signature (§8.4.2) of a constructor or method to a signature that has no parameterized types or type variables. The erasure of a constructor or method signature s is a signature consisting of the same name as s and the erasures of all the formal parameter types given in s .

The type parameters of a constructor or method (§8.4.4), and the return type (§8.4.5) of a method, also undergo erasure if the constructor or method's signature is erased.

The erasure of the signature of a generic method has no type parameters.

4.7 Reifiable Types

Because some type information is erased during compilation, not all types are available at run time. Types that are completely available at run time are known as *reifiable types*.

A type is *reifiable* if and only if one of the following holds:

- It refers to a non-generic class or interface type declaration.
- It is a parameterized type in which all type arguments are unbounded wildcards (§4.5.1).
- It is a raw type (§4.8).
- It is a primitive type (§4.2).
- It is an array type (§10.1) whose element type is reifiable.
- It is a nested type where, for each type T separated by ".", T itself is reifiable.

An intersection type is not reifiable.

4.8 Raw Types

To facilitate interfacing with non-generic legacy code, it is possible to use as a type the erasure (§4.6) of a parameterized type (§4.5). Such a type is called a *raw type*.

More precisely, a raw type is defined to be one of:

- The reference type that is formed by taking the name of a generic type declaration without an accompanying type argument list.
- An array type whose element type is a raw type.
- A non-static type member of a raw type *R* that is not inherited from a superclass or superinterface of *R*.

A non-generic class or interface type is not a raw type.

The superclasses (respectively, superinterfaces) of a raw type are the erasures of the superclasses (superinterfaces) of any of its parameterized invocations.

The type of a constructor (§8.8), instance method (§8.4, §9.4), or non-static field (§8.3) *M* of a raw type *C* that is not inherited from its superclasses or superinterfaces is the raw type that corresponds to the erasure of its type in the generic declaration corresponding to *C*.

The type of a static method or static field of a raw type *C* is the same as its type in the generic declaration corresponding to *C*.

It is a compile-time error to pass type arguments to a non-static type member of a raw type that is not inherited from its superclasses or superinterfaces.

It is a compile-time error to attempt to use a type member of a parameterized type as a raw type.

The use of raw types is allowed only as a concession to compatibility of legacy code. The use of raw types in code written after the introduction of genericity into the Java programming language is strongly discouraged. It is possible that future versions of the Java programming language will disallow the use of raw types.

To make sure that potential violations of the typing rules are always flagged, some accesses to members of a raw type will result in compile-time warnings. The rules for compile-time warnings when accessing members or constructors of raw types are as follows:

- At an assignment to a field: if the type of the left-hand operand is a raw type, then an unchecked warning occurs if erasure changes the field's type.

- At an invocation of a method or constructor: if the type of the class or interface to search (§15.12.1) is a raw type, then an unchecked warning occurs if erasure changes any of the types of any of the arguments to the method or constructor.
- No unchecked warning is required for a method call when the argument types do not change under erasure (even if the result type and/or throws clause changes), for reading from a field, or for a class instance creation of a raw type.

The supertype of a class may be a raw type. Member accesses for the class are treated as normal, and member accesses for the supertype are treated as for raw types. In the constructor of the class, calls to `super` are treated as method calls on a raw type.

4.9 Intersection Types

An intersection type takes the form $T_1 \& \dots \& T_n$ ($n > 0$), where T_i ($1 \leq i \leq n$) are type expressions.

Intersection types arise in the processes of capture conversion (§5.1.10) and type inference (§15.12.2.7). It is not possible to write an intersection type directly as part of a program; no syntax supports this.

The values of an intersection type are those objects that are values of all of the types T_i for $1 \leq i \leq n$.

The members of an intersection type $T_1 \& \dots \& T_n$ are determined as follows:

- For each T_i ($1 \leq i \leq n$), let C_i be the most specific class or array type such that $T_i <: C_i$. Then there must be some $T_k <: C_k$ such that $C_k <: C_i$ for any i ($1 \leq i \leq n$), or a compile-time error occurs.
- For $1 \leq j \leq n$, if T_j is a type variable, then let T_j' be an interface whose members are the same as the public members of T_j ; otherwise, if T_j is an interface, then let T_j' be T_j .
- Then the intersection type has the same members as a class type (Chapter 8, *Classes*) with an empty body, direct superclass C_k and direct superinterfaces $T_1', \dots, T_n'$, declared in the same package in which the intersection type appears.

4.10 Subtyping

The subtype and supertype relations are binary relations on types.

The *supertypes* of a type are obtained by reflexive and transitive closure over the direct supertype relation, written $S >_1 T$, which is defined by rules given later in this section. We write $S :> T$ to indicate that the supertype relation holds between S and T .

S is a *proper supertype* of T , written $S > T$, if $S :> T$ and $S \neq T$.

The *subtypes* of a type T are all types U such that T is a supertype of U , and the null type. We write $T <: S$ to indicate that the subtype relation holds between types T and S .

T is a *proper subtype* of S , written $T < S$, if $T <: S$ and $S \neq T$.

T is a *direct subtype* of S , written $T <_1 S$, if $S >_1 T$.

Subtyping does not extend through parameterized types: $T <: S$ does not imply that $C\langle T \rangle <: C\langle S \rangle$.

4.10.1 Subtyping among Primitive Types

The following rules define the direct supertype relation among the primitive types:

- `double` $>_1$ `float`
- `float` $>_1$ `long`
- `long` $>_1$ `int`
- `int` $>_1$ `char`
- `int` $>_1$ `short`
- `short` $>_1$ `byte`

4.10.2 Subtyping among Class and Interface Types

Given a generic type declaration $C\langle F_1, \dots, F_n \rangle$, the direct supertypes of the parameterized type $C\langle T_1, \dots, T_n \rangle$ are all of the following:

- the direct superclasses of C
- the direct superinterfaces of C
- the type `Object`, if C is an interface type with no direct superinterfaces.
- The raw type C .

The direct supertypes of the type $C\langle T_1, \dots, T_n \rangle$, where T_i ($1 \leq i \leq n$) is a type, are $D\langle U_1, \dots, U_k \rangle$, where:

- $D\langle U_1, \dots, U_k \rangle$ is a direct supertype of $C\langle F_1, \dots, F_n \rangle$, and θ is the substitution $[F_1 := T_1, \dots, F_n := T_n]$.
- $C\langle S_1, \dots, S_n \rangle$ where S_i contains T_i (§4.5.1) for $1 \leq i \leq n$.

The direct supertypes of the type $C\langle R_1, \dots, R_n \rangle$, where at least one of the R_i ($1 \leq i \leq n$) is a wildcard type argument, are the direct supertypes of $C\langle X_1, \dots, X_n \rangle$, where $C\langle X_1, \dots, X_n \rangle$ is the result of applying capture conversion (§5.1.10) to $C\langle R_1, \dots, R_n \rangle$.

The direct supertypes of an intersection type (§4.9) $T_1 \& \dots \& T_n$, are T_i ($1 \leq i \leq n$).

The direct supertypes of a type variable are the types listed in its bound.

A type variable is a direct supertype of its lower bound.

The direct supertypes of the null type are all reference types other than the null type itself.

4.10.3 Subtyping among Array Types

The following rules define the direct subtype relation among array types:

- If S and T are both reference types, then $S[] >_1 T[]$ iff $S >_1 T$.
- `Object` $>_1$ `Object[]`
- `Cloneable` $>_1$ `Object[]`
- `java.io.Serializable` $>_1$ `Object[]`
- If P is a primitive type, then:
 - ♦ `Object` $>_1$ $P[]$
 - ♦ `Cloneable` $>_1$ $P[]$
 - ♦ `java.io.Serializable` $>_1$ $P[]$

4.11 Where Types Are Used

Types are used when they appear in declarations or in certain expressions.

Types are also used as arguments to parameterized types.

4.12 Variables

A variable is a storage location and has an associated type, sometimes called its *compile-time type*, that is either a primitive type (§4.2) or a reference type (§4.3).

A variable's value is changed by an assignment (§15.26) or by a prefix or postfix + (increment) or -- (decrement) operator (§15.14.2, §15.14.3, §15.15.1, §15.15.2).

4.12.1 Variables of Primitive Type

A variable of a primitive type always holds a value of that exact primitive type.

4.12.2 Variables of Reference Type

A variable of a class type T can hold a null reference or a reference to an instance of class T or of any class that is a subclass of T .

A variable of an interface type can hold a null reference or a reference to any instance of any class that implements the interface.

If T is a primitive type, then a variable of type "array of T " can hold a null reference or a reference to any array of type "array of T ".

If T is a reference type, then a variable of type "array of T " can hold a null reference or a reference to any array of type "array of S " such that type S is a subclass or subinterface of type T .

A variable of type `Object[]` can hold an array of any reference type.

A variable of type `Object` can hold a null reference or a reference to any object, whether class instance or array.

It is possible that a variable of a parameterized type will refer to an object that is not of that parameterized type. This situation is known as *heap pollution*. Heap pollution can only occur if the program performed some operation involving a raw type that would give rise to an unchecked warning at compile-time (§4.9, §5.1.9).

The variable will always refer to an object that is an instance of a class that represents the parameterized type.

4.12.3 Kinds of Variables

There are seven kinds of variables:

1. A *class variable* is a field declared using the keyword `static` within a class declaration (§8.3.1.1), or with or without the keyword `static` within an interface declaration (§9.3).

A class variable is created when its class or interface is prepared (§12.3.2) and is initialized to a default value (§4.12.5). The class variable effectively ceases to exist when its class or interface is unloaded (§12.7).

2. An *instance variable* is a field declared within a class declaration without using the keyword `static` (§8.3.1.1).

If a class *T* has a field *a* that is an instance variable, then a new instance variable *a* is created and initialized to a default value (§4.12.5) as part of each newly created object of class *T* or of any class that is a subclass of *T* (§8.1.4). The instance variable effectively ceases to exist when the object of which it is a field is no longer referenced, after any necessary finalization of the object (§12.6) has been completed.

3. *Array components* are unnamed variables that are created and initialized to default values (§4.12.5) whenever a new object that is an array is created (Chapter 10, *Arrays*, §15.10). The array components effectively cease to exist when the array is no longer referenced.

4. *Method parameters* (§8.4.1) name argument values passed to a method.

For every parameter declared in a method declaration, a new parameter variable is created each time that method is invoked (§15.12). The new variable is initialized with the corresponding argument value from the method invocation. The method parameter effectively ceases to exist when the execution of the body of the method is complete.

5. *Constructor parameters* (§8.8.1) name argument values passed to a constructor.

For every parameter declared in a constructor declaration, a new parameter variable is created each time a class instance creation expression (§15.9) or explicit constructor invocation (§8.8.7) invokes that constructor. The new variable is initialized with the corresponding argument value from the creation expression or constructor invocation. The constructor parameter effectively ceases to exist when the execution of the body of the constructor is complete.

6. An *exception parameter* is created each time an exception is caught by a catch clause of a try statement (§14.20).

The new variable is initialized with the actual object associated with the exception (§11.3, §14.18). The exception parameter effectively ceases to exist when execution of the block associated with the catch clause is complete.

7. *Local variables* are declared by local variable declaration statements (§14.4).

Whenever the flow of control enters a block (§14.2) or for statement (§14.14), a new variable is created for each local variable declared in a local variable declaration statement immediately contained within that block or for statement.

A local variable declaration statement may contain an expression which initializes the variable. The local variable with an initializing expression is not initialized, however, until the local variable declaration statement that declares it is executed. (The rules of definite assignment (Chapter 16, *Definite Assignment*) prevent the value of a local variable from being used before it has been initialized or otherwise assigned a value.) The local variable effectively ceases to exist when the execution of the block or for statement is complete.

Were it not for one exceptional situation, a local variable could always be regarded as being created when its local variable declaration statement is executed. The exceptional situation involves the `switch` statement (§14.11), where it is possible for control to enter a block but bypass execution of a local variable declaration statement. Because of the restrictions imposed by the rules of definite assignment (Chapter 16, *Definite Assignment*), however, the local variable declared by such a bypassed local variable declaration statement cannot be used before it has been definitely assigned a value by an assignment expression (§15.26).

4.12.4 **final Variables**

A variable can be declared `final`. A final variable may only be assigned to once.

It is a compile-time error if a final variable is assigned to unless it is definitely unassigned (Chapter 16, *Definite Assignment*) immediately prior to the assignment.

A *blank final* is a final variable whose declaration lacks an initializer.

Once a `final` variable has been assigned, it always contains the same value. If a `final` variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object.

We call a variable, of primitive type or type `String`, that is `final` and initialized with a compile-time constant expression (§15.28) a *constant variable*.

4.12.5 Initial Values of Variables

Every variable in a program must have a value before its value is used.

- Each class variable, instance variable, or array component is initialized with a *default value* when it is created (§15.9, §15.10):
 - ♦ For type `byte`, the default value is zero, that is, the value of `(byte)0`.
 - ♦ For type `short`, the default value is zero, that is, the value of `(short)0`.
 - ♦ For type `int`, the default value is zero, that is, 0.
 - ♦ For type `long`, the default value is zero, that is, 0L.
 - ♦ For type `float`, the default value is positive zero, that is, `0.0f`.
 - ♦ For type `double`, the default value is positive zero, that is, `0.0d`.
 - ♦ For type `char`, the default value is the null character, that is, `'\u0000'`.
 - ♦ For type `boolean`, the default value is `false`.
 - ♦ For all reference types (§4.3), the default value is `null`.
- Each method parameter (§8.4.1) is initialized to the corresponding argument value provided by the invoker of the method (§15.12).
- Each constructor parameter (§8.8.1) is initialized to the corresponding argument value provided by a class instance creation expression (§15.9) or explicit constructor invocation (§8.8.7).
- An exception parameter (§14.20) is initialized to the thrown object representing the exception (§11.3, §14.18).
- A local variable (§14.4, §14.14) must be explicitly given a value before it is used, by either initialization (§14.4) or assignment (§15.26), in a way that can be verified by the Java compiler using the rules for definite assignment (Chapter 16, *Definite Assignment*).

4.12.6 Types, Classes, and Interfaces

In the Java programming language, every variable and every expression has a type that can be determined at compile-time. The type may be a primitive type or a reference type. Reference types include class types and interface types. Reference types are introduced by *type declarations*, which include class declarations (§8.1) and interface declarations (§9.1). We often use the term *type* to refer to either a class or an interface.

Every object belongs to some particular class: the class that was mentioned in the creation expression that produced the object, the class whose `Class` object was used to invoke a reflective method to produce the object, or the `String` class for objects implicitly created by the string concatenation operator `+` (§15.18.1). This class is called the *class of the object*. (Arrays also have a class, as described at the end of this section.) An object is said to be an instance of its class and of all superclasses of its class.

Sometimes a variable or expression is said to have a "run-time type". This refers to the class of the object referred to by the value of the variable or expression at run time, assuming that the value is not `null`.

The compile-time type of a variable is always declared, and the compile-time type of an expression can be deduced at compile-time. The compile-time type limits the possible values that the variable can hold or the expression can produce at run time. If a run-time value is a reference that is not `null`, it refers to an object or array that has a class, and that class will necessarily be compatible with the compile-time type.

Even though a variable or expression may have a compile-time type that is an interface type, there are no instances of interfaces. A variable or expression whose type is an interface type can reference any object whose class implements (§8.1.5) that interface.

Every array also has a class (§10.8); the method `getClass`, when invoked for an array object, will return a class object (of class `Class`) that represents the class of the array.

Conversions and Promotions

EVERY expression written in the Java programming language has a type that can be deduced from the structure of the expression and the types of the literals, variables, and methods mentioned in the expression. It is possible, however, to write an expression in a context where the type of the expression is not appropriate. In some cases, this leads to an error at compile time. In other cases, the context may be able to accept a type that is related to the type of the expression; as a convenience, rather than requiring the programmer to indicate a type conversion explicitly, the language performs an implicit *conversion* from the type of the expression to a type acceptable for its surrounding context.

A specific conversion from type *S* to type *T* allows an expression of type *S* to be treated at compile time as if it had type *T* instead. In some cases this will require a corresponding action at run time to check the validity of the conversion or to translate the run-time value of the expression into a form appropriate for the new type *T*.

In every conversion context, only certain specific conversions are permitted. For convenience of description, the specific conversions that are possible in the Java programming language are grouped into several broad categories:

- Identity conversions
- Widening primitive conversions
- Narrowing primitive conversions
- Widening reference conversions
- Narrowing reference conversions
- Boxing conversions
- Unboxing conversions
- Unchecked conversions

- Capture conversions
- String conversions
- Value set conversions

There are five *conversion contexts* in which conversion of expressions may occur. Each context allows conversions in some of the categories named above but not others. The term "conversion" is also used to describe the process of choosing a specific conversion for such a context. For example, we say that an expression that is an actual argument in a method invocation is subject to "method invocation conversion," meaning that a specific conversion will be implicitly chosen for that expression according to the rules for the method invocation argument context.

One conversion context is the operand of a numeric operator such as + or *. The conversion process for such operands is called *numeric promotion*. Promotion is special in that, in the case of binary operators, the conversion chosen for one operand may depend in part on the type of the other operand expression.

This chapter first describes the eleven categories of conversions (§5.1), including the special conversions to `String` allowed for the string concatenation operator +. Then the five conversion contexts are described:

- Assignment conversion (§5.2, §15.26) converts the type of an expression to the type of a specified variable. Assignment conversion may cause a `OutOfMemoryError` (as a result of boxing conversion (§5.1.7)), a `NullPointerException` (as a result of unboxing conversion (§5.1.8)), or a `ClassCastException` (as a result of an unchecked conversion (§5.1.9)) to be thrown at run time.
- Method invocation conversion (§5.3, §15.9, §15.12) is applied to each argument in a method or constructor invocation and, except in one case, performs the same conversions that assignment conversion does. Method invocation conversion may cause a `OutOfMemoryError` (as a result of boxing conversion (§5.1.7)), a `NullPointerException` (as a result of unboxing conversion (§5.1.8)), or a `ClassCastException` (as a result of an unchecked conversion (§5.1.9)) to be thrown at run time.
- Casting conversion (§5.5) converts the type of an expression to a type explicitly specified by a cast operator (§15.16). It is more inclusive than assignment or method invocation conversion, allowing any specific conversion other than a string conversion, but certain casts to a reference type may cause an exception at run time.

- String conversion (§5.4, §15.18.1) allows any type to be converted to type `String`.
- Numeric promotion (§5.6) brings the operands of a numeric operator to a common type so that an operation can be performed.

5.1 Kinds of Conversion

Specific type conversions in the Java programming language are divided into 13 categories.

5.1.1 Identity Conversions

A conversion from a type to that same type is permitted for any type.

5.1.2 Widening Primitive Conversion

19 specific conversions on primitive types are called the *widening primitive conversions*.

- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

A widening primitive conversion does not lose information about the overall magnitude of a numeric value, with the exception that a widening conversion from float to double that is not `strictfp` may lose information about the overall magnitude of the converted value.

A widening conversion from an integral type to another integral type, or from float to double in a `strictfp` expression, do not lose any information at all; the numeric value is preserved exactly.

A widening conversion of an int or a long value to float, or of a long value to double, may result in *loss of precision* - that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value

will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode (§4.2.4).

A widening conversion of a signed integer value to an integral type *T* simply sign-extends the two's-complement representation of the integer value to fill the wider format.

A widening conversion of a `char` to an integral type *T* zero-extends the representation of the `char` value to fill the wider format.

Despite the fact that loss of precision may occur, widening conversions among primitive types never result in a run-time exception (Chapter 11, *Exceptions*).

5.1.3 Narrowing Primitive Conversions

22 specific conversions on primitive types are called the narrowing primitive conversions.

- `short` to `byte` or `char`
- `char` to `byte` or `short`
- `int` to `byte`, `short`, or `char`
- `long` to `byte`, `short`, `char`, or `int`
- `float` to `byte`, `short`, `char`, `int`, or `long`
- `double` to `byte`, `short`, `char`, `int`, `long`, or `float`

A narrowing primitive conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range.

A narrowing primitive conversion from `double` to `float` is governed by the IEEE 754 rounding rules (§4.2.4). This conversion can lose precision, but also lose range, resulting in a `float` zero from a nonzero `double` and a `float` infinity from a finite `double`. A `double` NaN is converted to a `float` NaN and a `double` infinity is converted to the same-signed `float` infinity.

A narrowing conversion of a signed integer to an integral type *T* simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type *T*. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the sign of the resulting value to differ from the sign of the input value.

A narrowing conversion of a `char` to an integral type *T* likewise simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type *T*. In addition to a possible loss of information about the magnitude of the numeric

value, this may cause the resulting value to be a negative number, even though chars represent 16-bit unsigned integer values.

A narrowing conversion of a floating-point number to an integral type *T* takes two steps:

1. In the first step, the floating-point number is converted either to a `long`, if *T* is `long`, or to an `int`, if *T* is `byte`, `short`, `char`, or `int`, as follows:
 - If the floating-point number is NaN (§4.2.3), the result of the first step of the conversion is an `int` or `long` 0.
 - Otherwise, if the floating-point number is not an infinity, the floating-point value is rounded to an integer value *V*, rounding toward zero using IEEE 754 round-toward-zero mode (§4.2.3). Then there are two cases:
 - a. If *T* is `long`, and this integer value can be represented as a `long`, then the result of the first step is the `long` value *V*.
 - b. Otherwise, if this integer value can be represented as an `int`, then the result of the first step is the `int` value *V*.
 - Otherwise, one of the following two cases must be true:
 - a. The value must be too small (a negative value of large magnitude or negative infinity), and the result of the first step is the smallest representable value of type `int` or `long`.
 - b. The value must be too large (a positive value of large magnitude or positive infinity), and the result of the first step is the largest representable value of type `int` or `long`.
2. In the second step:
 - If *T* is `int` or `long`, the result of the conversion is the result of the first step.
 - If *T* is `byte`, `char`, or `short`, the result of the conversion is the result of a narrowing conversion to type *T* (§5.1.3) of the result of the first step.

Despite the fact that overflow, underflow, or other loss of information may occur, narrowing conversions among primitive types never result in a run-time exception (Chapter 11, *Exceptions*).

5.1.4 Widening and Narrowing Primitive Conversions

The following conversion combines both widening and narrowing primitive conversions:

- byte to char

First, the `byte` is converted to an `int` via widening primitive conversion (§5.1.2), and then the resulting `int` is converted to a `char` by narrowing primitive conversion (§5.1.3).

5.1.5 Widening Reference Conversions

A *widening reference* conversion exists from any reference type *S* to any reference type *T*, provided *S* is a subtype (§4.10) of *T*.

Widening reference conversions never require a special action at run time and therefore never throw an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

5.1.6 Narrowing Reference Conversions

Six kinds of conversions are called the *narrowing reference conversions*.

- From any reference type *S* to any reference type *T*, provided that *S* is a proper supertype (§4.10) of *T*.

An important special case is that there is a narrowing conversion from the class type `Object` to any other reference type.

- From any class type *C* to any non-parameterized interface type *K*, provided that *C* is not `final` and does not implement *K*.
- From any interface type *J* to any non-parameterized class type *C* that is not `final`.
- From any interface type *J* to any non-parameterized interface type *K*, provided that *J* is not a subinterface of *K*.
- From the interface types `Cloneable` and `java.io.Serializable` to any array type *T*[].
- From any array type *SC*[] to any array type *TC*[], provided that *SC* and *TC* are reference types and there is a narrowing reference conversion from *SC* to *TC*.

Such conversions require a test at run time to find out whether the actual reference value is a legitimate value of the new type. If not, then a `ClassCastException` is thrown.

5.1.7 Boxing Conversion

Boxing conversion converts expressions of primitive type to corresponding expressions of reference type. Specifically, the following eight conversions are called the *boxing conversions*:

- From type `boolean` to type `Boolean`
- From type `byte` to type `Byte`
- From type `char` to type `Character`
- From type `short` to type `Short`
- From type `int` to type `Integer`
- From type `long` to type `Long`
- From type `float` to type `Float`
- From type `double` to type `Double`

The null type may undergo boxing conversion (§15.25); the result is the null type.

At run time, boxing conversion proceeds as follows:

- If p is a value of type `boolean`, then boxing conversion converts p into a reference r of class and type `Boolean`, such that $r.\text{booleanValue}() == p$
- If p is a value of type `byte`, then boxing conversion converts p into a reference r of class and type `Byte`, such that $r.\text{byteValue}() == p$
- If p is a value of type `char`, then boxing conversion converts p into a reference r of class and type `Character`, such that $r.\text{charValue}() == p$
- If p is a value of type `short`, then boxing conversion converts p into a reference r of class and type `Short`, such that $r.\text{shortValue}() == p$
- If p is a value of type `int`, then boxing conversion converts p into a reference r of class and type `Integer`, such that $r.\text{intValue}() == p$
- If p is a value of type `long`, then boxing conversion converts p into a reference r of class and type `Long`, such that $r.\text{longValue}() == p$
- If p is a value of type `float` then:
 - ♦ If p is not NaN, then boxing conversion converts p into a reference r of class and type `Float`, such that $r.\text{floatValue}()$ evaluates to p
 - ♦ Otherwise, boxing conversion converts p into a reference r of class and type `Float` such that $r.\text{isNaN}()$ evaluates to `true`

- If p is a value of type `double`, then:
 - ♦ If p is not NaN, boxing conversion converts p into a reference r of class and type `Double`, such that `r.doubleValue()` evaluates to p
 - ♦ Otherwise, boxing conversion converts p into a reference r of class and type `Double` such that `r.isNaN()` evaluates to `true`
- If p is a value of any other type, boxing conversion is equivalent to an identity conversion §5.1.1

If the value p being boxed is `true`, `false`, a byte, or a char in the range `\u0000` to `\u007f`, or an `int` or short number between `-128` and `127`, then let r_1 and r_2 be the results of any two boxing conversions of p . It is always the case that $r_1 == r_2$.

A boxing conversion may result in an `OutOfMemoryError` if a new instance of one of the wrapper classes (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, or `Double`) needs to be allocated and insufficient storage is available.

5.1.8 Unboxing Conversion

Unboxing conversion converts expressions of reference type to corresponding expressions of primitive type. Specifically, the following eight conversions are called the *unboxing conversions*:

- From type `Boolean` to type `boolean`
- From type `Byte` to type `byte`
- From type `Character` to type `char`
- From type `Short` to type `short`
- From type `Integer` to type `int`
- From type `Long` to type `long`
- From type `Float` to type `float`
- From type `Double` to type `double`

At run time, unboxing conversion proceeds as follows:

- If r is a reference of type `Boolean`, then unboxing conversion converts r into `r.booleanValue()`
- If r is a reference of type `Byte`, then unboxing conversion converts r into `r.byteValue()`

- If r is a reference of type `Character`, then unboxing conversion converts r into $r.charValue()$
- If r is a reference of type `Short`, then unboxing conversion converts r into $r.shortValue()$
- If r is a reference of type `Integer`, then unboxing conversion converts r into $r.intValue()$
- If r is a reference of type `Long`, then unboxing conversion converts r into $r.longValue()$
- If r is a reference of type `Float`, unboxing conversion converts r into $r.floatValue()$
- If r is a reference of type `Double`, then unboxing conversion converts r into $r.doubleValue()$
- If r is `null`, unboxing conversion throws a `NullPointerException`

A type is said to be *convertible to a numeric type* if it is a numeric type (§4.2), or it is a reference type that may be converted to a numeric type by unboxing conversion.

A type is said to be *convertible to an integral type* if it is an integral type, or it is a reference type that may be converted to an integral type by unboxing conversion.

5.1.9 Unchecked Conversion

Let G name a generic type declaration with n type parameters.

There is an *unchecked conversion* from the raw class or interface type (§4.8) G to any parameterized type of the form $G<T_1, \dots, T_n>$.

There is an *unchecked conversion* from the raw array type $G[]$ to any parameterized type of the form $G[]<T_1, \dots, T_n>$.

Use of an unchecked conversion generates a compile-time *unchecked warning* unless the parameterized type $G<\dots>$ is a parameterized type in which all type arguments are unbounded wildcards (§4.5.1), or the unchecked warning is suppressed by the `SuppressWarnings` annotation (§9.6.3.5).

5.1.10 Capture Conversion

Let G name a generic type declaration with n type parameters $A_1, \dots, A_n$ with corresponding bounds $U_1, \dots, U_n$. There exists a *capture conversion* from $G<T_1, \dots, T_n>$ to $G<S_1, \dots, S_n>$, where, for $1 \leq i \leq n$:

- If T_i is a wildcard type argument (§4.5.1) of the form $?$, then S_i is a fresh type variable whose upper bound is $U_i[A_1:=S_1, \dots, A_n:=S_n]$ and whose lower bound is the null type.
- If T_i is a wildcard type argument of the form $? \text{ extends } B_i$, then S_i is a fresh type variable whose upper bound is $\text{glb}(B_i, U_i[A_1:=S_1, \dots, A_n:=S_n])$ and whose lower bound is the null type.

$\text{glb}(V_1, \dots, V_m)$ is $V_1 \& \dots \& V_m$. It is a compile-time error if, for any two classes (not interfaces) V_i and V_j , V_i is not a subclass of V_j or vice versa.

- If T_i is a wildcard type argument of the form $? \text{ super } B_i$, then S_i is a fresh type variable whose upper bound is $U_i[A_1:=S_1, \dots, A_n:=S_n]$ and whose lower bound is B_i .
- Otherwise, $S_i = T_i$.

Capture conversion on any type other than a parameterized type (§4.5) acts as an identity conversion (§5.1.1). Capture conversion never requires a special action at run time and therefore never throws an exception at run time.

Capture conversion is not applied recursively.

5.1.11 String Conversions

Any type may be converted to type `String` by *string conversion*.

A value x of primitive type T is first converted to a reference value as if by giving it as an argument to an appropriate class instance creation expression:

- If T is `boolean`, then use `new Boolean(x)`.
- If T is `char`, then use `new Character(x)`.
- If T is `byte`, `short`, or `int`, then use `new Integer(x)`.
- If T is `long`, then use `new Long(x)`.
- If T is `float`, then use `new Float(x)`.
- If T is `double`, then use `new Double(x)`.

This reference value is then converted to type `String` by string conversion.

Now only reference values need to be considered:

- If the reference is `null`, it is converted to the string `"null"` (four ASCII characters `n`, `u`, `l`, `l`).

- Otherwise, the conversion is performed as if by an invocation of the `toString` method of the referenced object with no arguments; but if the result of invoking the `toString` method is `null`, then the string `"null"` is used instead.

The `toString` method is defined by the primordial class `Object`; many classes override it, notably `Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double`, and `String`.

See §5.4 for details of the string conversion context.

5.1.12 Forbidden Conversions

Any conversion that is not explicitly allowed is forbidden.

5.1.13 Value Set Conversion

Value set conversion is the process of mapping a floating-point value from one value set to another without changing its type.

Within an expression that is not FP-strict (§15.4), value set conversion provides choices to an implementation of the Java programming language:

- If the value is an element of the float-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the float value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).
- If the value is an element of the double-extended-exponent value set, then the implementation may, at its option, map the value to the nearest element of the double value set. This conversion may result in overflow (in which case the value is replaced by an infinity of the same sign) or underflow (in which case the value may lose precision because it is replaced by a denormalized number or zero of the same sign).

Within an FP-strict expression (§15.4), value set conversion does not provide any choices; every implementation must behave in the same way:

- If the value is of type `float` and is not an element of the float value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.

- If the value is of type `double` and is not an element of the double value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

Within an FP-strict expression, mapping values from the float-extended-exponent value set or double-extended-exponent value set is necessary only when a method is invoked whose declaration is not FP-strict and the implementation has chosen to represent the result of the method invocation as an element of an extended-exponent value set.

Whether in FP-strict code or code that is not FP-strict, value set conversion always leaves unchanged any value whose type is neither `float` nor `double`.

5.2 Assignment Conversion

Assignment conversion occurs when the value of an expression is assigned (§15.26) to a variable: the type of the expression must be converted to the type of the variable.

Assignment contexts allow the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)
- a boxing conversion (§5.1.7) optionally followed by a widening reference conversion
- an unboxing conversion (§5.1.8) optionally followed by a widening primitive conversion.

If, after the conversions listed above have been applied, the resulting type is a raw type (§4.8), unchecked conversion (§5.1.9) may then be applied.

It is a compile-time error if the chain of conversions contains two parameterized types that are not in the subtype relation.

In addition, if the expression is a constant expression (§15.28) of type `byte`, `short`, `char`, or `int`:

- A narrowing primitive conversion may be used if the type of the variable is `byte`, `short`, or `char`, and the value of the constant expression is representable in the type of the variable.

- A narrowing primitive conversion followed by a boxing conversion may be used if the type of the variable is:
 - ♦ `Byte` and the value of the constant expression is representable in the type `byte`.
 - ♦ `Short` and the value of the constant expression is representable in the type `short`.
 - ♦ `Character` and the value of the constant expression is representable in the type `char`.

If the type of the expression cannot be converted to the type of the variable by a conversion permitted in an assignment context, then a compile-time error occurs.

If the type of the variable is `float` or `double`, then value set conversion is applied to the value *v* that is the result of the type conversion:

- If *v* is of type `float` and is an element of the float-extended-exponent value set, then the implementation must map *v* to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If *v* is of type `double` and is an element of the double-extended-exponent value set, then the implementation must map *v* to the nearest element of the double value set. This conversion may result in overflow or underflow.

If the type of an expression can be converted to the type of a variable by assignment conversion, we say the expression (or its value) is *assignable* to the variable or, equivalently, that the type of the expression is *assignment compatible* with the type of the variable.

If, after the type conversions above have been applied, the resulting value is an object which is not an instance of a subclass or subinterface of the erasure of the type of the variable, then a `ClassCastException` is thrown.

The only exceptions that an assignment conversion may cause are:

- An `OutOfMemoryError` as a result of a boxing conversion.
- A `ClassCastException` in the special circumstances indicated above.
- A `NullPointerException` as a result of an unboxing conversion on a null reference.

(Note, however, that an assignment may result in an exception in special cases involving array elements or field access - see §10.5 and §15.26.1.)

A value of the null type (the null reference is the only such value) may be assigned to any reference type, resulting in a null reference of that type.

5.3 Method Invocation Conversion

Method invocation conversion is applied to each argument value in a method or constructor invocation (§8.8.7.1, §15.9, §15.12): the type of the argument expression must be converted to the type of the corresponding parameter.

Method invocation contexts allow the use of one of the following:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a widening reference conversion (§5.1.5)
- a boxing conversion (§5.1.7) optionally followed by widening reference conversion
- an unboxing conversion (§5.1.8) optionally followed by a widening primitive conversion.

If, after the conversions listed above have been applied, the resulting type is a raw type (§4.8), an unchecked conversion (§5.1.9) may then be applied.

It is a compile-time error if the chain of conversions contains two parameterized types that are not in the subtype relation.

If the type of an argument expression is either `float` or `double`, then value set conversion (§5.1.13) is applied after the type conversion:

- If an argument value of type `float` is an element of the float-extended-exponent value set, then the implementation must map the value to the nearest element of the float value set. This conversion may result in overflow or underflow.
- If an argument value of type `double` is an element of the double-extended-exponent value set, then the implementation must map the value to the nearest element of the double value set. This conversion may result in overflow or underflow.

If, after the type conversions above have been applied, the resulting value is an object which is not an instance of a subclass or subinterface of the erasure of the corresponding formal parameter type, then a `ClassCastException` is thrown.

5.4 String Conversion

String conversion applies only to the operands of the binary `+` operator when one of the arguments is a `String`.

In this single special case, the other argument to the `+` undergoes string conversion (§5.1.11) to a `String`, and a new `String` which is the concatenation (§15.18.1) of the two strings is the result of the `+`.

5.5 Casting Conversion

Casting conversion is applied to the operand of a cast operator (§15.16): the type of the operand expression must be converted to the type explicitly named by the cast operator.

Casting contexts allow the use of one of:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- a narrowing primitive conversion (§5.1.3)
- a widening and narrowing primitive conversion (§5.1.4)
- a widening reference conversion (§5.1.5) optionally followed by either an unboxing conversion (§5.1.8) or an unchecked conversion (§5.1.9)
- a narrowing reference conversion (§5.1.6) optionally followed by either an unboxing conversion (§5.1.8) or an unchecked conversion
- a boxing conversion (§5.1.7) optionally followed by a widening reference conversion (§5.1.5)
- an unboxing conversion (§5.1.8) optionally followed by a widening primitive conversion (§5.1.2).

Value set conversion (§5.1.13) is applied after the type conversion.

The compile-time legality of a casting conversion is as follows:

- An expression of a primitive type can always undergo casting conversion to another primitive type without error, by identity conversion (if the types are the same) or by a widening primitive conversion or by a narrowing primitive conversion or by a widening and narrowing primitive conversion.

- An expression of a primitive type can always be undergo casting conversion to a reference type without error, by boxing conversion.
- An expression of a reference type can always undergo casting conversion to a primitive type without error, by unboxing conversion.
- An expression of a reference type can undergo casting conversion to another reference type if no compile-time error occurs given the rules in §5.5.1.

5.5.1 Reference Type Casting

Given a compile-time reference type S (source) and a compile-time reference type T (target), a casting conversion exists from S to T if no compile-time errors occur due to the following rules.

If S is a class type:

- If T is a class type, then either $|S| <: |T|$, or $|T| <: |S|$. Otherwise, a compile-time error occurs.

Furthermore, if there exists a supertype X of T , and a supertype Y of S , such that both X and Y are provably distinct parameterized types (§4.5), and that the erasures of X and Y are the same, a compile-time error occurs.

- If T is an interface type:
 - ♦ If S is not a `final` class (§8.1.1), then, if there exists a supertype X of T , and a supertype Y of S , such that both X and Y are provably distinct parameterized types, and that the erasures of X and Y are the same, a compile-time error occurs.
Otherwise, the cast is always legal at compile time (because even if S does not implement T , a subclass of S might).
 - ♦ If S is a `final` class (§8.1.1), then S must implement T , or a compile-time error occurs.
- If T is a type variable, then this algorithm is applied recursively, using the upper bound of T in place of T .
- If T is an array type, then S must be the class `Object`, or a compile-time error occurs.

If S is an interface type:

- If T is an array type, then S must be the type `java.io.Serializable` or `Cloneable` (the only interfaces implemented by arrays), or a compile-time error occurs.

- If T is a type that is not `final` (§8.1.1), then if there exists a supertype X of T , and a supertype Y of S , such that both X and Y are provably distinct parameterized types, and that the erasures of X and Y are the same, a compile-time error occurs.

Otherwise, the cast is always legal at compile time (because even if T does not implement S , a subclass of T might).

- If T is a type that is `final`, then:
 - ♦ If S is not a parameterized type or a raw type, then T must implement S , or a compile-time error occurs.
 - ♦ Otherwise, S is either a parameterized type that is an invocation of some generic type declaration G , or a raw type corresponding to a generic type declaration G . Then there must exist a supertype X of T , such that X is an invocation of G , or a compile-time error occurs.

Furthermore, if S and X are provably distinct parameterized types then a compile-time error occurs.

If S is a type variable, then this algorithm is applied recursively, using the upper bound of S in place of S .

If S is an intersection type $A_1 \& \dots \& A_n$, then it is a compile-time error if there exists an A_i ($1 \leq i \leq n$) such that S cannot be cast to A_i by this algorithm. That is, the success of the cast is determined by the most restrictive component of the intersection type.

If S is an array type $SC[]$, that is, an array of components of type SC :

- If T is a class type, then if T is not `Object`, then a compile-time error occurs (because `Object` is the only class type to which arrays can be assigned).
- If T is an interface type, then a compile-time error occurs unless T is the type `java.io.Serializable` or the type `Cloneable` (the only interfaces implemented by arrays).
- If T is a type variable, then:
 - ♦ If the upper bound of T is `Object` or `java.io.Serializable` or `Cloneable`, or a type variable that S could undergo casting conversion to, then the cast is legal (though unchecked).
 - ♦ If the upper bound of T is an array type $TC[]$, then a compile-time error occurs unless the type $SC[]$ can undergo casting conversion to $TC[]$.
 - ♦ Otherwise, a compile-time error occurs.

- If T is an array type $TC[]$, that is, an array of components of type TC , then a compile-time error occurs unless one of the following is true:
 - ♦ TC and SC are the same primitive type.
 - ♦ TC and SC are reference types and type SC can undergo casting conversion to TC .

5.5.2 Checked Casts and Unchecked Casts

A cast from a type S to a type T is *statically known to be correct* if and only if $S <: T$ (§4.10).

A cast from a type S to a parameterized type (§4.5) T is *unchecked* unless at least one of the following conditions holds:

- $S <: T$
- All of the type arguments (§4.5.1) of T are unbounded wildcards
- $T <: S$ and S has no subtype X other than T where the type arguments of X are not contained in the type arguments of T .

A cast from a type S to a type variable T is unchecked unless $S <: T$.

An unchecked cast from S to T is *completely unchecked* if the cast from $|S|$ to $|T|$ is statically known to be correct. Otherwise, it is *partially unchecked*.

An unchecked cast causes an unchecked warning to occur, unless it is suppressed using the `SuppressWarnings` annotation (§9.6.3.5).

A cast is a *checked* cast if it is not statically known to be correct and it is not unchecked.

If a cast to a reference type is not a compile-time error, there are several cases:

- The cast is statically known to be correct. No run time action is performed for such a cast.
- The cast is a completely unchecked cast. No run time action is performed for such a cast.
- The cast is a partially unchecked cast. Such a cast requires a run-time validity check. The check is performed as if the cast had been a checked cast between $|S|$ and $|T|$, as described below.
- The cast is a checked cast. Such a cast requires a run-time validity check. If the value at run time is `null`, then the cast is allowed. Otherwise, let R be the class of the object referred to by the run-time reference value, and let T be the erasure of the type named in the cast operator. A cast conversion must check, at run time,

that the class R is assignment compatible with the type T , via the algorithm in §5.5.3.

5.5.3 Checked Casts at Run-time

Here is the algorithm to check whether the run-time type R of an object is assignment compatible with the type T which is the erasure of the type named in the cast operator. If a run-time exception is thrown, it is a `ClassCastException`.

If R is an ordinary class (not an array class):

- If T is a class type, then R must be either the same class (§4.3.4) as T or a subclass of T , or a run-time exception is thrown.
- If T is an interface type, then R must implement (§8.1.5) interface T , or a run-time exception is thrown.
- If T is an array type, then a run-time exception is thrown.

If R is an interface:

- If T is a class type, then T must be `Object` (§4.3.2), or a run-time exception is thrown.
- If T is an interface type, then R must be either the same interface as T or a subinterface of T , or a run-time exception is thrown.
- If T is an array type, then a run-time exception is thrown.

If R is a class representing an array type $RC[]$, that is, an array of components of type RC :

- If T is a class type, then T must be `Object` (§4.3.2), or a run-time exception is thrown.
- If T is an interface type, then a run-time exception is thrown unless T is the type `java.io.Serializable` or the type `Cloneable` (the only interfaces implemented by arrays).
- If T is an array type $TC[]$, that is, an array of components of type TC , then a run-time exception is thrown unless one of the following is true:
 - ♦ TC and RC are the same primitive type.
 - ♦ TC and RC are reference types and type RC can be cast to TC by a recursive application of these run-time rules for casting.

5.6 Numeric Promotions

Numeric promotion is applied to the operands of an arithmetic operator.

Numeric promotion contexts allow the use of:

- an identity conversion (§5.1.1)
- a widening primitive conversion (§5.1.2)
- an unboxing conversion (§5.1.8)

Numeric promotions are used to convert the operands of a numeric operator to a common type so that an operation can be performed. The two kinds of numeric promotion are unary numeric promotion (§5.6.1) and binary numeric promotion (§5.6.2).

5.6.1 Unary Numeric Promotion

Some operators apply *unary numeric promotion* to a single operand, which must produce a value of a numeric type:

- If the operand is of compile-time type `Byte`, `Short`, `Character`, or `Integer`, it is subjected to unboxing conversion. The result is then promoted to a value of type `int` by a widening primitive conversion or an identity conversion.
- Otherwise, if the operand is of compile-time type `Long`, `Float`, or `Double` it is subjected to unboxing conversion.
- Otherwise, if the operand is of compile-time type `byte`, `short`, or `char`, unary numeric promotion promotes it to a value of type `int` by a widening primitive conversion.
- Otherwise, a unary numeric operand remains as is and is not converted.

In any case, value set conversion (§5.1.13) is then applied.

Unary numeric promotion is performed on expressions in the following situations:

- Each dimension expression in an array creation expression (§15.10)
- The index expression in an array access expression (§15.13)
- The operand of a unary plus operator `+` (§15.15.3)
- The operand of a unary minus operator `-` (§15.15.4)
- The operand of a bitwise complement operator `~` (§15.15.5)

- Each operand, separately, of a shift operator `>>`, `>>>`, or `<<` (§15.19).

A `long` shift distance (right operand) does not promote the value being shifted (left operand) to `long`.

5.6.2 Binary Numeric Promotion

When an operator applies *binary numeric promotion* to a pair of operands, each of which must denote a value that is convertible to a numeric type, the following rules apply, in order:

1. If any operand is of a reference type, it is subjected to unboxing conversion.
2. Widening primitive conversion is applied to convert either or both operands, as follows:
 - If either operand is of type `double`, the other is converted to `double`.
 - Otherwise, if either operand is of type `float`, the other is converted to `float`.
 - Otherwise, if either operand is of type `long`, the other is converted to `long`.
 - Otherwise, both operands are converted to type `int`.

After the type conversion, if any, value set conversion (§5.1.13) is applied to each operand.

Binary numeric promotion is performed on the operands of certain operators:

- The multiplicative operators `*`, `/` and `%` (§15.17)
- The addition and subtraction operators for numeric types `+` and `-` (§15.18.2)
- The numerical comparison operators `<`, `<=`, `>`, and `>=` (§15.20.1)
- The numerical equality operators `==` and `!=` (§15.21.1)
- The integer bitwise operators `&`, `^`, and `|` (§15.22.1)
- In certain cases, the conditional operator `? :` (§15.25)

Names

NAMES are used to refer to entities declared in a program. A declared entity (§6.1) is a package, class type (normal or enum), interface type (normal or annotation type), member (class, interface, field, or method) of a reference type, type parameter (of a class, interface, method or constructor), parameter (to a method, constructor, or exception handler), or local variable.

Names in programs are either *simple*, consisting of a single identifier, or *qualified*, consisting of a sequence of identifiers separated by "." tokens (§6.2).

Every declaration that introduces a name has a *scope* (§6.3), which is the part of the program text within which the declared entity can be referred to by a simple name.

A qualified name $N.x$ may be used to refer to a *member* of a package or reference type, where N is a simple or qualified name and x is an identifier. If N names a package, then x is a member of that package, which is either a class or interface type or a subpackage. If N names a reference type or a variable of a reference type, then x names a member of that type, which is either a class, an interface, a field, or a method.

In determining the meaning of a name (§6.5), the context of the occurrence is used to disambiguate among packages, types, variables, and methods with the same name.

Access control (§6.6) can be specified in a class, interface, method, or field declaration to control when *access* to a member is allowed. Access is a different concept from scope. Access specifies the part of the program text within which the declared entity can be referred to by a qualified name, a field access expression (§15.11), or a method invocation expression (§15.12) in which the method is not specified by a simple name. The default access is that a member can be accessed anywhere within the package that contains its declaration; other possibilities are `public`, `protected`, and `private`.

Fully qualified and canonical names (§6.7) are also discussed in this chapter.

6.1 Declarations

A *declaration* introduces an entity into a program and includes an identifier (§3.8) that can be used in a name to refer to this entity.

A declared entity is one of the following:

- A package, declared in a package declaration (§7.4)
- An imported type, declared in a single-type-import declaration (§7.5.1) or a type-import-on-demand declaration (§7.5.2)
- A class, declared in a class type declaration (§8.1)
- An interface, declared in an interface type declaration (§9.1)
- A type variable (§4.4), declared as a type parameter of a generic class (§8.1.2), interface (§9.1.2), method (§8.4.4) or constructor (§8.8.1).
- A member of a reference type (§8.2, §9.2, §10.7), one of the following:
 - ♦ A member class (§8.5, §9.5)
 - ♦ A member interface (§8.5, §9.5)
 - ♦ An enum constant (§8.9)
 - ♦ A field, one of the following:
 - ✧ A field declared in a class type (§8.3)
 - ✧ A field declared in an interface type (§9.3)
 - ✧ The field `length`, which is implicitly a member of every array type (§10.7)
 - ♦ A method, one of the following:
 - ✧ A method (`abstract` or otherwise) declared in a class type (§8.4)
 - ✧ A method (always `abstract`) declared in an interface type (§9.4)
- A parameter, one of the following:
 - ♦ A parameter of a method or constructor of a class (§8.4.1, §8.8.1)
 - ♦ A parameter of an abstract method of an interface (§9.4)
 - ♦ A parameter of an exception handler declared in a catch clause of a try statement (§14.20)
- A local variable, one of the following:
 - ♦ A local variable declared in a block (§14.4)

- ♦ A local variable declared in a `for` statement (§14.14)

Constructors (§8.8) are also introduced by declarations, but use the name of the class in which they are declared rather than introducing a new name.

6.2 Names and Identifiers

A *name* is used to refer to an entity declared in a program.

There are two forms of names: simple names and qualified names.

A *simple name* is a single identifier.

A *qualified name* consists of a name, a `"."` token, and an identifier.

In determining the meaning of a name (§6.5), the context in which the name appears is taken into account. The rules of §6.5 distinguish among contexts where a name must denote (refer to) a package (§6.5.3), a type (§6.5.5), a variable or value in an expression (§6.5.6), or a method (§6.5.7).

Not all identifiers in a program are a part of a name. Identifiers are also used in the following situations:

- In declarations (§6.1), where an identifier may occur to specify the name by which the declared entity will be known.
- As labels in labeled statements (§14.7) and in `break` and `continue` statements (§14.15, §14.16) that refer to statement labels.
- In field access expressions (§15.11), where an identifier occurs after a `"."` token to indicate a member of an object that is the value of an expression or the keyword `super` that appears before the `"."` token
- In some method invocation expressions (§15.12), where an identifier may occur after a `"."` token and before a `"("` token to indicate a method to be invoked for an object that is the value of an expression or the keyword `super` that appears before the `"."` token
- In qualified class instance creation expressions (§15.9), where an identifier occurs immediately to the right of the leftmost `new` token to indicate a type that must be a member of the compile-time type of the primary expression preceding the `"."` preceding the leftmost `new` token.

The identifiers used in labeled statements and their associated `break` and `continue` statements are completely separate from those used in declarations.

6.3 Scope of a Declaration

The *scope* of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name, provided it is visible (§6.4.1).

A declaration is said to be *in scope* at a particular point in a program if and only if the declaration's scope includes that point.

The scope of the declaration of an observable (§7.4.3) top level package is all observable compilation units (§7.3).

The declaration of a package that is not observable is never in scope.

The declaration of a subpackage is never in scope.

The package `java` is always in scope.

The scope of a type imported by a single-type-import declaration (§7.5.1) or a type-import-on-demand declaration (§7.5.2) is all the class and interface type declarations (§7.6) in the compilation unit in which the `import` declaration appears, as well as any annotations on the package declaration (if any) of the compilation unit .

The scope of a member imported by a single-static-import declaration (§7.5.3) or a static-import-on-demand declaration (§7.5.4) is all the class and interface type declarations (§7.6) in the compilation unit in which the `import` declaration appears, as well as any annotations on the package declaration (if any) of the compilation unit .

The scope of a top level type (§7.6) is all type declarations in the package in which the top level type is declared.

The scope of a member *m* declared in or inherited by a class type *C* (§8.1.6) is the entire body of *C*, including any nested type declarations.

The scope of a member *m* declared in or inherited by an interface type *I* (§9.1.4) is the entire body of *I*, including any nested type declarations.

The scope of a parameter of a method (§8.4.1) or constructor (§8.8.1) is the entire body of the method or constructor.

The scope of an class's type parameter (§8.1.2) is the type parameter section of the class declaration, the type parameter section of any superclass or superinterface of the class declaration, and the class body.

The scope of an interface's type parameter (§9.1.2) is the type parameter section of the interface declaration, the type parameter section of any superinterface of the interface declaration, and the interface body.

The scope of a method's type parameter (§8.4.4) is the entire declaration of the method, including the type parameter section, but excluding the method modifiers.

The scope of a constructor's type parameter (§8.8.4) is the entire declaration of the constructor, including the type parameter section, but excluding the constructor modifiers.

The scope of a local class immediately enclosed by a block (§14.2) is the rest of the immediately enclosing block, including its own class declaration.

The scope of a local class in a switch block statement group (§14.11) is the rest of the immediately enclosing switch block statement group, including its own class declaration.

The scope of a local variable declaration in a block (§14.4.2) is the rest of the block in which the declaration appears, starting with its own initializer (§14.4) and including any further declarators to the right in the local variable declaration statement.

The scope of a local variable declared in the *ForInit* part of a basic for statement (§14.14.1) includes all of the following:

- Its own initializer
- Any further declarators to the right in the *ForInit* part of the for statement
- The *Expression* and *ForUpdate* parts of the for statement
- The contained *Statement*

The scope of a local variable declared in the *FormalParameter* part of an enhanced for statement (§14.14.2) is the contained *Statement*.

The scope of a parameter of an exception handler that is declared in a catch clause of a try statement (§14.20) is the entire block associated with the catch.

The scope of an enum constant *C* declared in an enum type *T* is the body of *T*, and any case label of a switch statement whose expression is of enum type *T*.

These rules imply that declarations of class and interface types need not appear before uses of the types.

6.4 Shadowing and Obscuring

6.4.1 Shadowing

Some declarations may be *shadowed* in part of their scope by another declaration of the same name, in which case a simple name cannot be used to refer to the declared entity.

A declaration d of a type named n shadows the declarations of any other types named n that are in scope at the point where d occurs throughout the scope of d .

A declaration d of a field, method parameter, constructor parameter, or exception handler parameter named n shadows the declarations of any other fields, method parameters, constructor parameters, or exception handler parameters named n that are in scope at the point where d occurs throughout the scope of d .

A declaration d of a local variable named n shadows the declarations of any fields named n that are in scope at the point where d occurs throughout the scope of d (§14.4.3).

A declaration d of a method named n shadows the declarations of any other methods named n that are in an enclosing scope at the point where d occurs throughout the scope of d .

A package declaration never shadows any other declaration.

A type-import-on-demand declaration never causes any other declaration to be shadowed.

A static-import-on-demand declaration never causes any other declaration to be shadowed.

A single-type-import declaration d in a compilation unit c of package p that imports a type named n shadows, throughout c , the declarations of:

- any top level type named n declared in another compilation unit of p
- any type named n imported by a type-import-on-demand declaration in c
- any type named n imported by a static-import-on-demand declaration in c

A single-static-import declaration d in a compilation unit c of package p that imports a field named n shadows the declaration of any static field named n imported by a static-import-on-demand declaration in c , throughout c .

A single-static-import declaration d in a compilation unit c of package p that imports a method named n with signature s shadows the declaration of any

static method named n with signature s imported by a static-import-on-demand declaration in c , throughout c .

A single-static-import declaration d in a compilation unit c of package p that imports a type named n shadows, throughout c , the declarations of:

- any static type named n imported by a static-import-on-demand declaration in c ;
- any top level type (§7.6) named n declared in another compilation unit (§7.3) of p ;
- any type named n imported by a type-import-on-demand declaration (§7.5.2) in c .

A declaration d is said to be *visible at point p in a program* if the scope of d includes p , and d is not shadowed by any other declaration at p .

When the program point we are discussing is clear from context, we will often simply say that a declaration is *visible*.

6.4.2 Obscuring

A simple name may occur in contexts where it may potentially be interpreted as the name of a variable, a type, or a package. In these situations, the rules of §6.5 specify that a variable will be chosen in preference to a type, and that a type will be chosen in preference to a package. Thus, it is may sometimes be impossible to refer to a visible type or package declaration via its simple name. We say that such a declaration is *obscured*.

-
-
-
-
-
-
-
-

6.5 Determining the Meaning of a Name

The meaning of a name depends on the context in which it is used. The determination of the meaning of a name requires three steps:

1. First, context causes a name syntactically to fall into one of six categories: *PackageName*, *TypeName*, *ExpressionName*, *MethodName*, *PackageOrTypeName*, or *AmbiguousName*.
2. Second, a name that is initially classified by its context as an *AmbiguousName* or as a *PackageOrTypeName* is then reclassified to be a *PackageName*, *TypeName*, or *ExpressionName*.
3. Third, the resulting category then dictates the final determination of the meaning of the name (or a compilation error if the name has no meaning).

PackageName:

Identifier

PackageName . Identifier

TypeName:

Identifier

PackageOrTypeName . Identifier

ExpressionName:

Identifier

AmbiguousName . Identifier

MethodName:

Identifier

AmbiguousName . Identifier

PackageOrTypeName:

Identifier

PackageOrTypeName . Identifier

AmbiguousName:

Identifier

AmbiguousName . Identifier

The name of a field, parameter, or local variable may be used as an expression (§15.14.1).

The name of a method may appear in an expression only as part of a method invocation expression (§15.12).

The name of a class or interface type may appear in an expression only as part of a class literal (§15.8.2), a qualified `this` expression (§15.8.4), a class instance creation expression (§15.9), an array creation expression (§15.10), a cast expression (§15.16), an `instanceof` expression (§15.20.2), an enum constant (§8.9), or as part of a qualified name for a field or method.

The name of a package may appear in an expression only as part of a qualified name for a class or interface type.

6.5.1 Syntactic Classification of a Name According to Context

A name is syntactically classified as a *PackageName* in these contexts:

- In a package declaration (§7.4)
- To the left of the "." in a qualified *PackageName*

A name is syntactically classified as a *TypeName* in these contexts:

- In a single-type-import declaration (§7.5.1)
- To the left of the "." in a single static import (§7.5.3) declaration
- To the left of the "." in a static import-on-demand (§7.5.4) declaration
- To the left of the "<" in a parameterized type (§4.5)
- In a type argument list of a parameterized type
- In an explicit type argument list in a method or constructor invocation
- In an `extends` clause in a type variable declaration (§8.1.2)
- In an `extends` clause of a wildcard type argument (§4.5.1)
- In a `super` clause of a wildcard type argument (§4.5.1)
- In an `extends` clause in a class declaration (§8.1.4)
- In an `implements` clause in a class declaration (§8.1.5)
- In an `extends` clause in an interface declaration (§9.1.3)
- After the "@" sign in an annotation (§9.7)
- As a *Type* (or the part of a *Type* that remains after all brackets are deleted) in any of the following contexts:

- ♦ In a field declaration (§8.3, §9.3)
- ♦ As the result type of a method (§8.4, §9.4)
- ♦ As the type of a formal parameter of a method or constructor (§8.4.1, §8.8.1, §9.4)
- ♦ As the type of an exception that can be thrown by a method or constructor (§8.4.6, §8.8.5, §9.4)
- ♦ As the type of a local variable (§14.4)
- ♦ As the type of an exception parameter in a catch clause of a try statement (§14.20)
- ♦ As the type in a class literal (§15.8.2)
- ♦ As the qualifying type of a qualified this expression (§15.8.4).
- ♦ As the class type which is to be instantiated in an unqualified class instance creation expression (§15.9)
- ♦ As the direct superclass or direct superinterface of an anonymous class (§15.9.5) which is to be instantiated in an unqualified class instance creation expression (§15.9)
- ♦ As the element type of an array to be created in an array creation expression (§15.10)
- ♦ As the qualifying type of field access using the keyword super (§15.11.2)
- ♦ As the qualifying type of a method invocation using the keyword super (§15.12)
- ♦ As the type mentioned in the cast operator of a cast expression (§15.16)
- ♦ As the type that follows the instanceof relational operator (§15.20.2)

A name is syntactically classified as an *ExpressionName* in these contexts:

- As the qualifying expression in a qualified superclass constructor invocation (§8.8.7.1)
- As the qualifying expression in a qualified class instance creation expression (§15.9)
- As the array reference expression in an array access expression (§15.13)
- As a *PostfixExpression* (§15.14)
- As the left-hand operand of an assignment operator (§15.26)

A name is syntactically classified as a *MethodName* in these contexts:

- Before the "(" in a method invocation expression (§15.12)
- To the left of the "=" sign in an annotation's element value pair (§9.7)

A name is syntactically classified as a *PackageOrTypeName* in these contexts:

- To the left of the "." in a qualified *TypeName*
- In a type-import-on-demand declaration (§7.5.2)

A name is syntactically classified as an *AmbiguousName* in these contexts:

- To the left of the "." in a qualified *ExpressionName*
- To the left of the "." in a qualified *MethodName*
- To the left of the "." in a qualified *AmbiguousName*
- In the default value clause of an annotation type element declaration (§9.6)
- To the right of an "=" in an element value pair (§9.7)

6.5.2 Reclassification of Contextually Ambiguous Names

An *AmbiguousName* is then reclassified as follows.

If the *AmbiguousName* is a simple name, consisting of a single *Identifier*:

- If the *Identifier* appears within the scope (§6.3) of a local variable declaration (§14.4) or parameter declaration (§8.4.1, §8.8.1, §14.20) or field declaration (§8.3) with that name, then the *AmbiguousName* is reclassified as an *ExpressionName*.
- Otherwise, if a field of that name is declared in the compilation unit (§7.3) containing the *Identifier* by a single-static-import declaration (§7.5.3), or by a static-import-on-demand declaration (§7.5.4) then the *AmbiguousName* is reclassified as an *ExpressionName*.
- Otherwise, if the *Identifier* appears within the scope (§6.3) of a top level class (Chapter 8, *Classes*) or interface type declaration (Chapter 9, *Interfaces*), a local class declaration (§14.3) or member type declaration (§8.5, §9.5) with that name, then the *AmbiguousName* is reclassified as a *TypeName*.
- Otherwise, if a type of that name is declared in the compilation unit (§7.3) containing the *Identifier*, either by a single-type-import declaration (§7.5.1), or by a type-import-on-demand declaration (§7.5.2), or by a single-static-import

declaration (§7.5.3), or by a static-import-on-demand declaration (§7.5.4), then the *AmbiguousName* is reclassified as a *TypeName*.

- Otherwise, the *AmbiguousName* is reclassified as a *PackageName*. A later step determines whether or not a package of that name actually exists.

If the *AmbiguousName* is a qualified name, consisting of a name, a ".", and an *Identifier*, then the name to the left of the "." is first reclassified, for it is itself an *AmbiguousName*. There is then a choice:

- If the name to the left of the "." is reclassified as a *PackageName*, then if there is a package whose name is the name to the left of the "." and that package contains a declaration of a type whose name is the same as the *Identifier*, then this *AmbiguousName* is reclassified as a *TypeName*.

Otherwise, this *AmbiguousName* is reclassified as a *PackageName*. A later step determines whether or not a package of that name actually exists.

- If the name to the left of the "." is reclassified as a *TypeName*, then if the *Identifier* is the name of a method or field of the type denoted by *TypeName*, this *AmbiguousName* is reclassified as an *ExpressionName*.

Otherwise, if the *Identifier* is the name of a member type of the type denoted by *TypeName*, this *AmbiguousName* is reclassified as a *TypeName*. Otherwise, a compile-time error occurs.

- If the name to the left of the "." is reclassified as an *ExpressionName*, then let *T* be the type of the expression denoted by *ExpressionName*. If the *Identifier* is the name of a method or field of the type denoted by *T*, this *AmbiguousName* is reclassified as an *ExpressionName*.

Otherwise, if the *Identifier* is the name of a member type (§8.5, §9.5) of the type denoted by *T*, then this *AmbiguousName* is reclassified as a *TypeName*. Otherwise, a compile-time error occurs.

6.5.3 Meaning of Package Names

The meaning of a name classified as a *PackageName* is determined as follows.

6.5.3.1 Simple Package Names

If a package name consists of a single *Identifier*, then this identifier denotes a top level package named by that identifier. If no top level package of that name is in scope (§6.3), then a compile-time error occurs.

6.5.3.2 *Qualified Package Names*

If a package name is of the form *Q.Id*, then *Q* must also be a package name. The package name *Q.Id* names a package that is the member named *Id* within the package named by *Q*.

If *Q* does not name an observable package (§7.4.3), or *Id* is not the simple name of an observable subpackage of that package, then a compile-time error occurs.

6.5.4 **Meaning of *PackageOrTypeNames***

6.5.4.1 *Simple PackageOrTypeNames*

If the *PackageOrTypeName*, *Q*, occurs in the scope of a type named *Q*, then the *PackageOrTypeName* is reclassified as a *TypeName*.

Otherwise, the *PackageOrTypeName* is reclassified as a *PackageName*. The meaning of the *PackageOrTypeName* is the meaning of the reclassified name.

6.5.4.2 *Qualified PackageOrTypeNames*

Given a qualified *PackageOrTypeName* of the form *Q.Id*, if the type or package denoted by *Q* has a member type named *Id*, then the qualified *PackageOrTypeName* name is reclassified as a *TypeName*.

Otherwise, it is reclassified as a *PackageName*. The meaning of the qualified *PackageOrTypeName* is the meaning of the reclassified name.

6.5.5 **Meaning of Type Names**

The meaning of a name classified as a *TypeName* is determined as follows.

6.5.5.1 *Simple Type Names*

If a type name consists of a single *Identifier*, then the identifier must occur in the scope of exactly one visible declaration of a type with this name, or a compile-time error occurs. The meaning of the type name is that type.

6.5.5.2 *Qualified Type Names*

If a type name is of the form *Q.Id*, then *Q* must be either a type name or a package name.

If *Id* names exactly one accessible (§6.6) type that is a member of the type or package denoted by *Q*, then the qualified type name denotes that type.

If *Id* does not name a member type (§8.5, §9.5) within *Q*, or there is not exactly one accessible (§6.6) member type named *Id* within *Q*, or *Id* names a static member type (§8.5.2) within *Q* and *Q* is parameterized, then a compile-time error occurs.

6.5.6 Meaning of Expression Names

The meaning of a name classified as an *ExpressionName* is determined as follows.

6.5.6.1 Simple Expression Names

If an expression name consists of a single *Identifier*, then there must be exactly one visible declaration denoting either a local variable, parameter, or field in scope at the point at which the *Identifier* occurs. Otherwise, a compile-time error occurs.

If the declaration declares a `final` variable which is definitely assigned before the simple expression, the meaning of the name is the value of that variable. Otherwise, the meaning of the expression name is the variable declared by the declaration.

If the field is an instance variable (§8.3), the expression name must appear within the declaration of an instance method (§8.4), constructor (§8.8), instance initializer (§8.6), or instance variable initializer (§8.3.2.2). If it appears within a static method (§8.4.3.2), static initializer (§8.7), or initializer for a static variable (§8.3.2.1, §12.4.2), then a compile-time error occurs.

If the expression name appears in a context where it is subject to assignment conversion or method invocation conversion or casting conversion, then the type of the expression name is the declared type of the field, local variable, or parameter after capture conversion (§5.1.10). Otherwise, the type of the expression name is the declared type of the field, local variable or parameter.

6.5.6.2 Qualified Expression Names

If an expression name is of the form *Q.Id*, then *Q* has already been classified as a package name, a type name, or an expression name.

If *Q* is a package name, then a compile-time error occurs.

If *Q* is a type name that names a class type (Chapter 8, *Classes*), then:

- If there is not exactly one accessible (§6.6) member of the class type that is a field named *Id*, then a compile-time error occurs.

- Otherwise, if the single accessible member field is not a class variable (that is, it is not declared `static`), then a compile-time error occurs.
- Otherwise, if the class variable is declared `final`, then $Q.Id$ denotes the value of the class variable. The type of the expression $Q.Id$ is the declared type of the class variable after capture conversion (§5.1.10).

If $Q.Id$ appears in a context that requires a variable and not a value, then a compile-time error occurs.

- Otherwise, $Q.Id$ denotes the class variable. The type of the expression $Q.Id$ is the declared type of the class variable after capture conversion (§5.1.10).

If Q is a type name that names an interface type (Chapter 9, *Interfaces*), then:

- If there is not exactly one accessible (§6.6) member of the interface type that is a field named Id , then a compile-time error occurs.
- Otherwise, $Q.Id$ denotes the value of the field. The type of the expression $Q.Id$ is the declared type of the field after capture conversion (§5.1.10).

If $Q.Id$ appears in a context that requires a variable and not a value, then a compile-time error occurs.

If Q is an expression name, let T be the type of the expression Q :

- If T is not a reference type, a compile-time error occurs.
- If there is not exactly one accessible (§6.6) member of the type T that is a field named Id , then a compile-time error occurs.
- Otherwise, if this field is any of the following:
 - ♦ A field of an interface type
 - ♦ A `final` field of a class type (which may be either a class variable or an instance variable)
 - ♦ The `final` field length of an array type

then $Q.Id$ denotes the value of the field, unless it appears in a context that requires a variable and the field is a definitely unassigned blank `final` field, in which case it yields a variable. The type of the expression $Q.Id$ is the declared type of the field after capture conversion (§5.1.10).

If $Q.Id$ appears in a context that requires a variable and not a value, and the field denoted by $Q.Id$ is definitely assigned, then a compile-time error occurs.

- Otherwise, $Q.Id$ denotes a variable, the field Id of class T , which may be either a class variable or an instance variable. The type of the expression $Q.Id$ is the type of the field member after capture conversion (§5.1.10).

Note that expression names may be qualified by type names, but not by types in general. A consequence is that it is not possible to access a class variable through a parameterized type.

6.5.7 Meaning of Method Names

A *MethodName* can appear only in a method invocation expression (§15.12) or as an element name in an element-value pair (§9.7). The meaning of a name classified as a *MethodName* is determined as follows.

6.5.7.1 Simple Method Names

A simple method name may appear as the element name in an element-value pair. The *Identifier* in an *ElementValuePair* must be the simple name of one of the elements of the annotation type identified by *TypeName* in the containing annotation. Otherwise, a compile-time error occurs. (In other words, the identifier in an element-value pair must also be a method name in the interface identified by *TypeName*.)

Otherwise, a simple method name necessarily appears in the context of a method invocation expression. In that case, if a method name consists of a single *Identifier*, then *Identifier* is the method name to be used for method invocation. The *Identifier* must name at least one visible (§6.4.1) method that is in scope at the point where the *Identifier* appears or a method imported by a single-static-import declaration (§7.5.3) or static-import-on-demand declaration (§7.5.4) within the compilation unit within which the *Identifier* appears.

6.5.7.2 Qualified Method Names

A qualified method name can only appear in the context of a method invocation expression.

If a method name is of the form $Q.Id$, then Q has already been classified as a package name, a type name, or an expression name.

If Q is a package name, then a compile-time error occurs. Otherwise, Id is the method name to be used for method invocation.

If Q is a type name, then Id must name at least one `static` method of the type Q .

If Q is an expression name, then let T be the type of the expression Q . Id must name at least one method of the type T .

Like expression names, method names may be qualified by type names, but not by types in general. The implications are similar to those for expression names as discussed in §6.5.6.2.

6.6 Access Control

The Java programming language provides mechanisms for *access control*, to prevent the users of a package or class from depending on unnecessary details of the implementation of that package or class. If access is permitted, then the accessed entity is said to be *accessible*.

Qualified names are a means of access to members of packages and reference types. When the name of such a member is classified from its context (§6.5.1) as a qualified type name (denoting a member of a package or reference type, §6.5.5.2) or a qualified expression name (denoting a member of a reference type, §6.5.6.2), access control is applied.

6.6.1 Determining Accessibility

- A package is always accessible.
- If a class or interface type is declared `public`, then it may be accessed by any code, provided that the compilation unit (§7.3) in which it is declared is observable. If a top level class or interface type is not declared `public`, then it may be accessed only from within the package in which it is declared.
- An array type is accessible if and only if its element type is accessible.
- A member (class, interface, field, or method) of a reference (class, interface, or array) type or a constructor of a class type is accessible only if the type is accessible and the member or constructor is declared to permit access:
 - ♦ If the member or constructor is declared `public`, then access is permitted. All members of interfaces are implicitly `public`.
 - ♦ Otherwise, if the member or constructor is declared `protected`, then access is permitted only when one of the following is true:

- ✧ Access to the member or constructor occurs from within the package containing the class in which the protected member or constructor is declared.
- ✧ Access is correct as described in §6.6.2.
- ◆ Otherwise, if the member or constructor is declared *private*, then access is permitted if and only if it occurs within the body of the top level class (§7.6) that encloses the declaration of the member or constructor.
- ◆ Otherwise, we say there is default access, which is permitted only when the access occurs from within the package in which the type is declared.

6.6.2 Details on protected Access

A protected member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object.

6.6.2.1 Access to a protected Member

Let *C* be the class in which a protected member is declared. Access is permitted only within the body of a subclass *S* of *C*.

In addition, if *Id* denotes an instance field or instance method, then:

- If the access is by a qualified name *Q.Id*, where *Q* is an *ExpressionName*, then the access is permitted if and only if the type of the expression *Q* is *S* or a subclass of *S*.
- If the access is by a field access expression *E.Id*, where *E* is a *Primary* expression, or by a method invocation expression *E.Id*(. . .), where *E* is a *Primary* expression, then the access is permitted if and only if the type of *E* is *S* or a subclass of *S*.

6.6.2.2 Qualified Access to a protected Constructor

Let *C* be the class in which a protected constructor is declared and let *S* be the innermost class in whose declaration the use of the protected constructor occurs. Then:

- If the access is by a superclass constructor invocation *super*(. . .) or by a qualified superclass constructor invocation of the form *E.super*(. . .), where *E* is a *Primary* expression, then the access is permitted.

- If the access is by an anonymous class instance creation expression of the form `new C(. . .){...}` or by a qualified class instance creation expression of the form `E.new C(. . .){...}`, where *E* is a *Primary* expression, then the access is permitted.
- Otherwise, if the access is by a simple class instance creation expression of the form `new C(. . .)` or by a qualified class instance creation expression of the form `E.new C(. . .)`, where *E* is a *Primary* expression, then the access is not permitted.

A `protected` constructor can be accessed by a class instance creation expression (that does not declare an anonymous class) only from within the package in which it is defined.

6.7 Fully Qualified Names and Canonical Names

Every named package, top level class, top level interface, and primitive type has a *fully qualified name*.

- The fully qualified name of a primitive type is the keyword for that primitive type, namely `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, or `double`.
- The fully qualified name of a named package that is not a subpackage of a named package is its simple name.
- The fully qualified name of a named package that is a subpackage of another named package consists of the fully qualified name of the containing package, followed by `"."`, followed by the simple (member) name of the subpackage.
- The fully qualified name of a top level class or top level interface that is declared in an unnamed package is the simple name of the class or interface.
- The fully qualified name of a top level class or top level interface that is declared in a named package consists of the fully qualified name of the package, followed by `"."`, followed by the simple name of the class or interface.

Each member class, member interface, and array type may have a fully qualified name.

- A member class or member interface *M* of another class *C* has a fully qualified name if and only if *C* has a fully qualified name.

In that case, the fully qualified name of *M* consists of the fully qualified name of *C*, followed by `"."`, followed by the simple name of *M*.

- An array type has a fully qualified name if and only if its element type has a fully qualified name.

In that case, the fully qualified name of an array type consists of the fully qualified name of the component type of the array type followed by "[]".

A local class does not have a fully qualified name.

Every named package, top level class, top level interface, and primitive type has a *canonical name*.

For every named package, top level class, top level interface, and primitive type, the canonical name is the same as the fully qualified name.

Each member class, member interface, and array type may have a canonical name.

- A member class or member interface M declared in another class C has a canonical name if and only if C has a canonical name.

In that case, the canonical name of M consists of the canonical name of C , followed by ".", followed by the simple name of M .

- An array type has a canonical name if and only if its element type has a canonical name.

In that case, the canonical name of the array type consists of the canonical name of the component type of the array type followed by "[]".

A local class does not have a canonical name.

Packages

PROGRAMS are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts. A top level type is accessible (§6.6) outside the package that declares it only if the type is declared `public`.

The naming structure for packages is hierarchical (§7.1). The members of a package are class and interface types (§7.6), which are declared in compilation units of the package, and subpackages, which may contain compilation units and subpackages of their own.

A package can be stored in a file system or in a database (§7.2). Packages that are stored in a file system may have certain constraints on the organization of their compilation units to allow a simple implementation to find classes easily.

A package consists of a number of compilation units (§7.3). A compilation unit automatically has access to all types declared in its package and also automatically imports all of the `public` types declared in the predefined package `java.lang`.

For small programs and casual development, a package can be unnamed (§7.4.2) or have a simple name, but if code is to be widely distributed, unique package names should be chosen using qualified names. This can prevent the conflicts that would otherwise occur if two development groups happened to pick the same package name and these packages were later to be used in a single program.

7.1 Package Members

The members of a package are its subpackages and all the top level class types (§7.6, Chapter 8, *Classes*) and top level interface types (Chapter 9, *Interfaces*) declared in all the compilation units (§7.3) of the package.

If the fully qualified name (§6.7) of a package is P , and Q is a subpackage of P , then $P.Q$ is the fully qualified name of the subpackage, and furthermore denotes a package.

A package may not contain two members of the same name, or a compile-time error results.

The hierarchical naming structure for packages is intended to be convenient for organizing related packages in a conventional manner, but has no significance in itself other than the prohibition against a package having a subpackage with the same simple name as a top level type (§7.6) declared in that package.

There is no special access relationship between a package named `oliver` and another package named `oliver.twist`, or between packages named `evelyn.wood` and `evelyn.waugh`. That is, the code in a package named `oliver.twist` has no better access to the types declared within package `oliver` than code in any other package.

7.2 Host Support for Packages

Each host system determines how packages and compilation units are created and stored.

In simple implementations of the Java SE platform, packages and compilation units may be stored in a local file system. Other implementations may store them using a distributed file system or some form of database.

If a host system stores packages and compilation units in a database, then the database must not impose the optional restrictions (§7.6) on compilation units permissible in file-based implementations.

Systems that use a database must, however, provide an option to convert a program to a form that obeys the restrictions, for purposes of export to file-based implementations.

7.3 Compilation Units

CompilationUnit is the goal symbol (§2.1) for the syntactic grammar (§2.3) of Java programs. It is defined by the following productions:

CompilationUnit:

PackageDeclaration_{opt} ImportDeclarations_{opt} TypeDeclarations_{opt}

ImportDeclarations:

ImportDeclaration

ImportDeclarations ImportDeclaration

TypeDeclarations:

TypeDeclaration

TypeDeclarations TypeDeclaration

A *compilation unit* consists of three parts, each of which is optional:

- A package declaration (§7.4), giving the fully qualified name (§6.7) of the package to which the compilation unit belongs.

A compilation unit that has no package declaration is part of an unnamed package (§7.4.2).

- `import` declarations (§7.5) that allow types from other packages and static members of types to be referred to using their simple names
- Top level type declarations (§7.6) of class and interface types

Every compilation unit implicitly imports every `public` type name declared in the predefined package `java.lang`, as if the declaration `import java.lang.*;` appeared at the beginning of each compilation unit immediately after any package statement. As a result, the names of all those types are available as simple names in every compilation unit.

Types declared in different compilation units can depend on each other, circularly. A Java compiler must arrange to compile all such types at the same time.

All the compilation units of the predefined package `java` and its subpackages `lang` and `io` are always *observable*.

For all other packages, the host system determines which compilation units are *observable*.

7.4 Package Declarations

A package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs.

7.4.1 Named Packages

A *package declaration* in a compilation unit specifies the name (§6.2) of the package to which the compilation unit belongs.

PackageDeclaration:
Annotations_{opt} package PackageName ;

The package name mentioned in a package declaration must be the fully qualified name (§6.7) of the package.

The *PackageName* in a package declaration ensures there is an observable package with the supplied canonical name, and that it is not subject to the rules in §6.5.3 for determining the meaning of a package name.

The keyword `package` may optionally be preceded by annotation modifiers. If an annotation *a* (§9.7) on a package declaration corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.PACKAGE`, or a compile-time error occurs.

At most one annotated package declaration is permitted for a given package.

7.4.2 Unnamed Packages

A compilation unit that has no package declaration is part of an *unnamed package*.

Unnamed packages are provided by the Java SE platform principally for convenience when developing small or temporary applications or when just beginning development.

Note that an unnamed package cannot have subpackages, since the syntax of a package declaration always includes a reference to a named top level package.

An implementation of the Java SE platform must support at least one unnamed package; it may support more than one unnamed package but is not required to do so. Which compilation units are in each unnamed package is determined by the host system.

7.4.3 Observability of a Package

A package is *observable* if and only if either:

- A compilation unit containing a declaration of the package is observable.
- A subpackage of the package is observable.

The packages `java`, `java.lang`, and `java.io` are always observable.

7.5 Import Declarations

An *import declaration* allows a named type or a static member to be referred to by a simple name (§6.2) that consists of a single identifier.

Without the use of an appropriate `import` declaration, the only way to refer to a type declared in another package, or a static member of another type, is to use a fully qualified name (§6.7).

ImportDeclaration:

SingleTypeImportDeclaration

TypeImportOnDemandDeclaration

SingleStaticImportDeclaration

StaticImportOnDemandDeclaration

A single-type-import declaration (§7.5.1) imports a single named type, by mentioning its canonical name (§6.7).

A type-import-on-demand declaration (§7.5.2) imports all the accessible (§6.6) types of a named type or named package as needed, by mentioning the canonical name of a type or package.

A single static import declaration (§7.5.3) imports all accessible static members with a given name from a type, by giving its canonical name.

A static-import-on-demand declaration (§7.5.4) imports all accessible static members of a named type as needed, by mentioning the canonical name of a type.

An `import` declaration makes types or members available by their simple names only within the compilation unit that actually contains the `import` declaration. The scope of the type(s) or member(s) introduced by an `import` declaration specifically does not include the *PackageName* of a package declaration, other `import` declarations in the current compilation unit, or other compilation units in the same package.

7.5.1 Single-Type-Import Declaration

A *single-type-import declaration* imports a single type by giving its canonical name (§6.7), making it available under a simple name in the class and interface

declarations of the compilation unit in which the single-type-import declaration appears.

SingleTypeImportDeclaration:
`import TypeName ;`

The *TypeName* must be the canonical name of a class type, interface type, enum type, or annotation type.

It is a compile-time error if the named type is not accessible (§6.6).

If two single-type-import declarations in the same compilation unit attempt to import types with the same simple name, then a compile-time error occurs, unless the two types are the same type, in which case the duplicate declaration is ignored.

If the type imported by the the single-type-import declaration is declared in the compilation unit that contains the `import` declaration, the `import` declaration is ignored.

If a compilation unit contains both a single-static-import declaration (§7.5.3) that imports a type whose simple name is *n*, and a single-type-import declaration (§7.5.1) that imports a type whose simple name is *n*, a compile-time error occurs.

If another top level type with the same simple name is otherwise declared in the current compilation unit except by a type-import-on-demand declaration (§7.5.2) or a static-import-on-demand declaration (§7.5.4), then a compile-time error occurs.

7.5.2 Type-Import-on-Demand Declaration

A *type-import-on-demand declaration* allows all accessible (§6.6) types declared in the type or package named by a canonical name to be imported as needed.

TypeImportOnDemandDeclaration:
`import PackageOrTypeName . * ;`

The *PackageOrTypeName* must be the canonical name of a package, a class type, an interface type, an enum type, or an annotation type.

It is a compile-time error if the named package or type is not accessible (§6.6).

Two or more type-import-on-demand declarations in the same compilation unit may name the same type or package. All but one of these declarations are considered redundant; the effect is as if that type was imported only once.

If a compilation unit contains both a static-import-on-demand declaration and a type-import-on-demand (§7.5.2) declaration that name the same type, the effect is as if the `static` member types of that type were imported only once.

It is not a compile-time error to name the current package or `java.lang` in a type-import-on-demand declaration. The type-import-on-demand declaration is ignored in such cases.

7.5.3 Single Static Import Declaration

A *single-static-import declaration* imports all accessible (§6.6) `static` members with a given simple name from a type. This makes these `static` members available under their simple name in the class and interface declarations of the compilation unit in which the single-static import declaration appears.

SingleStaticImportDeclaration:

```
import static TypeName . Identifier ;
```

The *TypeName* must be the canonical name (§6.7) of a class type, interface type, enum type, or annotation type.

It is a compile-time error if the named type is not accessible (§6.6).

The *Identifier* must name at least one `static` member of the named type. It is a compile-time error if there is no `static` member of that name, or if all of the named members are not accessible.

It is permissible for one single-static-import declaration to import several fields or types with the same name, or several methods with the same name and signature.

If a compilation unit contains both a single-static-import (§7.5.3) declaration that imports a type whose simple name is *n*, and a single-type-import declaration (§7.5.1) that imports a type whose simple name is *n*, a compile-time error occurs.

If a single-static-import declaration imports a type whose simple name is *n*, and the compilation unit also declares a top level type (§7.6) whose simple name is *n*, a compile-time error occurs.

7.5.4 Static-Import-on-Demand Declaration

A *static-import-on-demand declaration* allows all accessible (§6.6) `static` members of the type named by a canonical name to be imported as needed.

StaticImportOnDemandDeclaration:

```
import static TypeName . * ;
```

The *TypeName* must be the canonical name of a class type, interface type, enum type, or annotation type.

It is a compile-time error if the named type is not accessible.

Two or more static-import-on-demand declarations in the same compilation unit may name the same type ; the effect is as if there was exactly one such declaration.

Two or more static-import-on-demand declarations in the same compilation unit may name the same member; the effect is as if the member was imported exactly once.

Note that it is permissible for one static-import-on-demand declaration to import several fields or types with the same name, or several methods with the same name and signature.

If a compilation unit contains both a static-import-on-demand declaration and a type-import-on-demand (§7.5.2) declaration that name the same type, the effect is as if the static member types of that type were imported only once.

A static-import-on-demand declaration never causes any other declaration to be shadowed.

7.6 Top Level Type Declarations

A *top level type declaration* declares a top level class type (Chapter 8, *Classes*) or a top level interface type (Chapter 9, *Interfaces*).

```
TypeDeclaration:
  ClassDeclaration
  InterfaceDeclaration
  ;
```

By default, the top level types declared in a package are accessible only within the compilation units of that package, but a type may be declared to be `public` to grant access to the type from code in other packages (§6.6, §8.1.1, §9.1.1).

It is a compile-time error if a top level type declaration contains any one of the following access modifiers: `protected`, `private`, or `static`.

If a top level type named *T* is declared in a compilation unit of a package whose fully qualified name is *P*, then the fully qualified name of the type is *P. T*.

If the type is declared in an unnamed package (§7.4.2), then the type has the fully qualified name *T*.

An implementation of the Java SE platform must keep track of types within packages by their binary names (§13.1). Multiple ways of naming a type must be expanded to binary names to make sure that such names are understood as referring to the same type.

If and only if packages are stored in a file system (§7.2), the host system may choose to enforce the restriction that it is a compile-time error if a type is not found in a file under a name composed of the type name plus an extension (such as `.java` or `.jav`) if either of the following is true:

- The type is referred to by code in other compilation units of the package in which the type is declared.
- The type is declared `public` (and therefore is potentially accessible from code in other packages).

It is a compile-time error if the name of a top level type appears as the name of any other top level class or interface type declared in the same package.

It is a compile-time error if the name of a top level type is also declared as a type by a single-type-import declaration (§7.5.1) in the compilation unit (§7.3) containing the type declaration.

Classes

CLASS declarations define new reference types and describe how they are implemented (§8.1).

A *top level class* is a class that is not a nested class.

A *nested class* is any class whose declaration occurs within the body of another class or interface.

This chapter discusses the common semantics of all classes - top level (§7.6) and nested (including member classes (§8.5, §9.5), local classes (§14.3) and anonymous classes (§15.9.5)). Details that are specific to particular kinds of classes are discussed in the sections dedicated to these constructs.

A named class may be declared `abstract` (§8.1.1.1) and must be declared abstract if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses. A class may be declared `final` (§8.1.1.2), in which case it cannot have subclasses. If a class is declared `public`, then it can be referred to from other packages. Each class except `Object` is an extension of (that is, a subclass of) a single existing class (§8.1.4) and may implement interfaces (§8.1.5). Classes may be *generic*, that is, they may declare type variables whose bindings may differ among different instances of the class.

Classes may be decorated with annotations (§9.7) just like any other kind of declaration.

The body of a class declares members (fields and methods and nested classes and interfaces), instance and static initializers, and constructors (§8.1.6). The scope (§6.3) of a member (§8.2) is the entire body of the declaration of the class to which the member belongs. Field, method, member class, member interface, and constructor declarations may include the access modifiers (§6.6) `public`, `protected`, or `private`. The members of a class include both declared and inherited members (§8.2). Newly declared fields can hide fields declared in a superclass or superinterface. Newly declared class members and interface members

can hide class or interface members declared in a superclass or superinterface. Newly declared methods can hide, implement, or override methods declared in a superclass or superinterface.

Field declarations (§8.3) describe class variables, which are incarnated once, and instance variables, which are freshly incarnated for each instance of the class. A field may be declared `final` (§8.3.1.2), in which case it can be assigned to only once. Any field declaration may include an initializer.

Member class declarations (§8.5) describe nested classes that are members of the surrounding class. Member classes may be `static`, in which case they have no access to the instance variables of the surrounding class; or they may be inner classes (§8.1.3).

Member interface declarations (§8.5) describe nested interfaces that are members of the surrounding class.

Method declarations (§8.4) describe code that may be invoked by method invocation expressions (§15.12). A class method is invoked relative to the class type; an instance method is invoked with respect to some particular object that is an instance of a class type. A method whose declaration does not indicate how it is implemented must be declared `abstract`. A method may be declared `final` (§8.4.3.3), in which case it cannot be hidden or overridden. A method may be implemented by platform-dependent native code (§8.4.3.4). A `synchronized` method (§8.4.3.6) automatically locks an object before executing its body and automatically unlocks the object on return, as if by use of a `synchronized` statement (§14.19), thus allowing its activities to be synchronized with those of other threads (Chapter 17, *Threads and Locks*).

Method names may be overloaded (§8.4.9).

Instance initializers (§8.6) are blocks of executable code that may be used to help initialize an instance when it is created (§15.9).

Static initializers (§8.7) are blocks of executable code that may be used to help initialize a class.

Constructors (§8.8) are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded (§8.8.8).

8.1 Class Declaration

A class declaration specifies a new named reference type.

There are two kinds of class declarations: *normal class declarations* and *enum declarations*.

ClassDeclaration:

NormalClassDeclaration

EnumDeclaration

NormalClassDeclaration:

*ClassModifiers*_{opt} **class** *Identifier* *TypeParameters*_{opt}
*Super*_{opt} *Interfaces*_{opt} *ClassBody*

The rules in this section apply to all class declarations unless this specification explicitly states otherwise. In many cases, special restrictions apply to enum declarations. Enum declarations are described in detail in §8.9.

The *Identifier* in a class declaration specifies the name of the class.

It is a compile-time error if a class has the same simple name as any of its enclosing classes or interfaces.

8.1.1 Class Modifiers

A class declaration may include *class modifiers*.

ClassModifiers:

ClassModifier

ClassModifiers *ClassModifier*

ClassModifier: one of

Annotation **public** **protected** **private**

abstract **static** **final** **strictfp**

If an annotation *a* (§9.7) on a class declaration corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.TYPE`, or a compile-time error occurs.

The access modifier **public** (§6.6) pertains only to top level classes (§7.6) and to member classes (§8.5), not to local or anonymous classes.

The access modifiers `protected` and `private` (§6.6) pertain only to member classes within a directly enclosing class or enum declaration (§8.5.1).

The modifier `static` pertains only to member classes (§8.5.2), not to top level or local or anonymous classes.

It is a compile-time error if the same modifier appears more than once in a class declaration.

8.1.1.1 abstract *Classes*

An abstract class is a class that is incomplete, or to be considered incomplete.

Normal classes may have abstract methods (§8.4.3.1, §9.4), that is, methods that are declared but not yet implemented, only if they are abstract classes. If a normal class that is not abstract contains an abstract method, then a compile-time error occurs.

Enum types (§8.9) must not be declared abstract; doing so will result in a compile-time error.

It is a compile-time error for an enum type *E* to have an abstract method *m* as a member unless *E* has one or more enum constants, and all of *E*'s enum constants have class bodies that provide concrete implementations of *m*.

It is a compile-time error for the class body of an enum constant to declare an abstract method.

A class *C* has abstract methods if any of the following is true:

- *C* explicitly contains a declaration of an abstract method (§8.4.3).
- Any of *C*'s superclasses has an abstract method and *C* neither declares nor inherits a method that implements (§8.4.8.1) it.
- A direct superinterface (§8.1.5) of *C* declares or inherits a method (which is therefore necessarily abstract) and *C* neither declares nor inherits a method that implements it.

It is a compile-time error if an attempt is made to create an instance of an abstract class using a class instance creation expression (§15.9).

A subclass of an abstract class that is not itself abstract may be instantiated, resulting in the execution of a constructor for the abstract class and, therefore, the execution of the field initializers for instance variables of that class.

It is a compile-time error to declare an abstract class type such that it is not possible to create a subclass that implements all of its abstract methods.

8.1.1.2 `final` Classes

A class can be declared `final` if its definition is complete and no subclasses are desired or required.

It is a compile-time error if the name of a `final` class appears in the `extends` clause (§8.1.4) of another class declaration; this implies that a `final` class cannot have any subclasses.

It is a compile-time error if a class is declared both `final` and `abstract`, because the implementation of such a class could never be completed (§8.1.1.1).

Because a `final` class never has any subclasses, the methods of a `final` class are never overridden (§8.4.8.1).

8.1.1.3 `strictfp` Classes

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the class declaration (including within instance variable initializers, instance initializers, static initializers, and constructors) be explicitly FP-strict (§15.4).

This implies that all methods declared in the class, and all nested types declared in the class, are implicitly `strictfp`.

8.1.2 Generic Classes and Type Parameters

A class is *generic* if it declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the class. The type parameter section follows the class name and is delimited by angle brackets.

TypeParameters:

`< TypeParameterList >`

TypeParameterList:

`TypeParameterList , TypeParameter`

`TypeParameter`

In a class's type parameter section, a type variable *T* *directly depends* on a type variable *S* if *S* is the bound of *T*, while *T depends* on *S* if either *T* directly depends on *S* or *T* directly depends on a type variable *U* that depends on *S* (using this definition recursively). It is a compile-time error if a type variable in a class's type parameter section depends on itself.

A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section. All of these parameterized types share the same class at runtime.

It is a compile-time error if a generic class is a direct or indirect subclass of `Throwable`.

It is a compile-time error to refer to a type parameter of a class *C* anywhere in the declaration of a static member of *C* or the declaration of a static member of any type declaration nested within *C*.

It is a compile-time error to refer to a type parameter of a class *C* within the static initializer of *C* or any class nested within *C*.

Parameterized class declarations can be nested inside other declarations.

8.1.3 Inner Classes and Enclosing Instances

An *inner class* is a nested class that is not explicitly or implicitly declared `static`.

Inner classes include local (§14.3), anonymous (§15.9.5) and non-static member classes (§8.5).

Inner classes may not declare static initializers (§8.7) or member interfaces.

Inner classes may not declare static members, unless they are constant variables (§4.12.4).

Inner classes may inherit static members that are not compile-time constants even though they may not declare them. Nested classes that are not inner classes may declare static members freely, in accordance with the usual rules of the Java programming language.

Member interfaces (§8.5) are implicitly `static` so they are never considered to be inner classes.

A statement or expression *occurs in a static context* if and only if the innermost method, constructor, instance initializer, static initializer, field initializer, or explicit constructor invocation statement enclosing the statement or expression is a static method, a static initializer, the variable initializer of a static variable, or an explicit constructor invocation statement (§8.8.7).

An inner class *C* is a *direct inner class of a class O* if *O* is the immediately lexically enclosing class of *C* and the declaration of *C* does not occur in a static context.

A class *C* is an *inner class of class O* if it is either a direct inner class of *O* or an inner class of an inner class of *O*.

A class *O* is the *zeroth lexically enclosing class of itself*.

A class *O* is the *n'th lexically enclosing class of a class C* if it is the immediately enclosing class of the *n-1'th* lexically enclosing class of *C*.

An instance *i* of a direct inner class *C* of a class *O* is associated with an instance of *O*, known as the *immediately enclosing instance* of *i*. The immediately enclosing instance of an object, if any, is determined when the object is created (§15.9.2).

An object *o* is the *zeroth lexically enclosing instance of itself*.

An object *o* is the *n'th lexically enclosing instance of an instance i* if it is the immediately enclosing instance of the *n-1'th* lexically enclosing instance of *i*.

When an inner class refers to an instance variable that is a member of a lexically enclosing class, the variable of the corresponding lexically enclosing instance is used.

A blank `final` (§4.12.4) field of a lexically enclosing class may not be assigned within an inner class.

An instance of an inner class *I* whose declaration occurs in a static context has no lexically enclosing instances. However, if *I* is immediately declared within a static method or static initializer then *I* does have an *enclosing block*, which is the innermost block statement lexically enclosing the declaration of *I*.

For every superclass *S* of *C* which is itself a direct inner class of a class *SO*, there is an instance of *SO* associated with *i*, known as the *immediately enclosing instance of i with respect to S*. The immediately enclosing instance of an object with respect to its class' direct superclass, if any, is determined when the superclass constructor is invoked via an explicit constructor invocation statement.

Any local variable, formal method parameter, or exception handler parameter used but not declared in an inner class must be declared `final`.

Any local variable used but not declared in an inner class must be definitely assigned (Chapter 16, *Definite Assignment*) before the body of the inner class.

Inner classes whose declarations do not occur in a static context may freely refer to the instance variables of their enclosing class. An instance variable is always defined with respect to an instance. In the case of instance variables of an enclosing class, the instance variable must be defined with respect to an enclosing instance of that class.

8.1.4 Superclasses and Subclasses

The optional *extends* clause in a normal class declaration specifies the *direct superclass* of the current class.

Super:
extends *ClassType*

A class is said to be a *direct subclass* of its direct superclass. The direct superclass is the class from whose implementation the implementation of the current class is derived.

The direct superclass of an enum type *E* is *Enum<E>*.

The *extends* clause must not appear in the definition of the class *Object*, because it is the primordial class and has no direct superclass.

Given a (possibly generic) class declaration for $C\langle F_1, \dots, F_n \rangle$ ($n \geq 0$, $C \neq \text{Object}$), the *direct superclass* of the class type (§4.5) $C\langle F_1, \dots, F_n \rangle$ is the type given in the *extends* clause of the declaration of *C* if an *extends* clause is present, or *Object* otherwise.

Let $C\langle F_1, \dots, F_n \rangle$ ($n > 0$) be a generic class declaration. The direct superclass of the parameterized class type $C\langle T_1, \dots, T_n \rangle$, where T_i ($1 \leq i \leq n$) is a type, is $D\langle U_1 \theta, \dots, U_k \theta \rangle$, where $D\langle U_1, \dots, U_k \rangle$ is the direct superclass of $C\langle F_1, \dots, F_n \rangle$, and θ is the substitution $[F_1 := T_1, \dots, F_n := T_n]$.

The *ClassType* must name an accessible (§6.6) class type, or a compile-time error occurs.

If the specified *ClassType* names a class that is *final* (§8.1.1.2), then a compile-time error occurs, as *final* classes are not allowed to have subclasses.

It is a compile-time error if the *ClassType* names the class *Enum* or any invocation of it.

If the *TypeName* is followed by any type arguments, it must be a correct invocation of the type declaration denoted by *TypeName*, and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.

The *subclass* relationship is the transitive closure of the direct subclass relationship. A class *A* is a subclass of class *C* if either of the following is true:

- *A* is the direct subclass of *C*
- There exists a class *B* such that *A* is a subclass of *B*, and *B* is a subclass of *C*, applying this definition recursively.

Class *C* is said to be a *superclass* of class *A* whenever *A* is a subclass of *C*.

A class *C* *directly depends* on a type *T* if *T* is mentioned in the `extends` or `implements` clause of *C* either as a superclass or superinterface, or as a qualifier of a superclass or superinterface name.

A class *C* *depends* on a reference type *T* if any of the following conditions hold:

- *C* directly depends on *T*.
- *C* directly depends on an interface *I* that depends (§9.1.3) on *T*.
- *C* directly depends on a class *D* that depends on *T* (using this definition recursively).

It is a compile-time error if a class depends on itself.

If circularly declared classes are detected at run time, as classes are loaded (§12.2), then a `ClassCircularityError` is thrown.

8.1.5 Superinterfaces

The optional `implements` clause in a class declaration lists the names of interfaces that are direct superinterfaces of the class being declared.

Interfaces:

`implements` *InterfaceTypeList*

InterfaceTypeList:

InterfaceType

InterfaceTypeList , *InterfaceType*

Given a (possibly generic) class declaration for $C\langle F_1, \dots, F_n \rangle$ ($n \geq 0$, $C \neq \text{Object}$), the *direct superinterfaces* of the class type (§4.5) $C\langle F_1, \dots, F_n \rangle$ are the types given in the `implements` clause of the declaration of *C*, if an `implements` clause is present.

Let $C\langle F_1, \dots, F_n \rangle$ ($n > 0$) be a generic class declaration. The direct superinterfaces of the parameterized class type $C\langle T_1, \dots, T_n \rangle$, where T_i ($1 \leq i \leq n$) is a type, are all types $I\langle U_1 \ \theta, \dots, U_k \ \theta \rangle$, where $I\langle U_1, \dots, U_k \rangle$ is a direct superinterface of $C\langle F_1, \dots, F_n \rangle$, and θ is the substitution $[F_1 := T_1, \dots, F_n := T_n]$.

Each *InterfaceType* must name an accessible (§6.6) interface type, or a compile-time error occurs.

If the *TypeName* is followed by any type arguments, it must be a correct invocation of the type declaration denoted by *TypeName*, and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.

It is a compile-time error if the same interface is mentioned as a direct superinterface two or more times in a single `implements` clause's names. This is true even if the interface is named in different ways.

An interface type *I* is a superinterface of class type *C* if any of the following is true:

- *I* is a direct superinterface of *C*.
- *C* has some direct superinterface *J* for which *I* is a superinterface, using the definition of "superinterface of an interface" given in §9.1.3.
- *I* is a superinterface of the direct superclass of *C*.

A class is said to *implement* all its superinterfaces.

A class can have a superinterface in more than one way.

Unless the class being declared is `abstract`, the declarations of all the method members of each direct superinterface must be implemented either by a declaration in this class or by an existing method declaration inherited from the direct superclass, because a class that is not abstract is not permitted to have abstract methods (§8.1.1.1).

It is permitted for a single method declaration in a class to implement methods of more than one superinterface.

A class may not at the same time be a subtype of two interface types which are different invocations of the same generic interface (§9.1.2), or an invocation of a generic interface and a raw type naming that same generic interface.

8.1.6 Class Body and Member Declarations

A *class body* may contain declarations of members of the class, that is, fields (§8.3), methods (§8.4), classes (§8.5), and interfaces (§8.5).

A class body may also contain instance initializers (§8.6), static initializers (§8.7), and declarations of constructors (§8.8) for the class.

```
ClassBody:
  { ClassBodyDeclarationsopt }

ClassBodyDeclarations:
  ClassBodyDeclaration
  ClassBodyDeclarations ClassBodyDeclaration

ClassBodyDeclaration:
  ClassMemberDeclaration
  InstanceInitializer
  StaticInitializer
  ConstructorDeclaration

ClassMemberDeclaration:
  FieldDeclaration
  MethodDeclaration
  ClassDeclaration
  InterfaceDeclaration
  ;
```

8.2 Class Members

The members of a class type are all of the following:

- Members inherited from its direct superclass (§8.1.4), except in class `Object`, which has no direct superclass
- Members inherited from any direct superinterfaces (§8.1.5)
- Members declared in the body of the class (§8.1.6)

Members of a class that are declared `private` are not inherited by subclasses of that class.

Only members of a class that are declared `protected` or `public` are inherited by subclasses declared in a package other than the one in which the class is declared.

Constructors, static initializers, and instance initializers are not members and therefore are not inherited.

We use the phrase *the type of a member* to denote:

- For a field, its type.

- For a method, an ordered 3-tuple consisting of:
 - ♦ argument types: a list of the types of the arguments to the method member.
 - ♦ return type: the return type of the method member.
 - ♦ throws clause: exception types declared in the throws clause of the method member.

Fields, methods, and member types of a class type may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5). However, this is discouraged as a matter of style.

8.3 Field Declarations

The variables of a class type are introduced by *field declarations*.

FieldDeclaration:

*FieldModifiers*_{opt} *Type* *VariableDeclarators* ;

VariableDeclarators:

VariableDeclarator

VariableDeclarators , *VariableDeclarator*

VariableDeclarator:

VariableDeclaratorId

VariableDeclaratorId = *VariableInitializer*

VariableDeclaratorId:

Identifier

VariableDeclaratorId []

VariableInitializer:

Expression

ArrayInitializer

The *Identifier* in a *FieldDeclarator* may be used in a name to refer to the field.

More than one field may be declared in a single field declaration by using more than one declarator; the *FieldModifiers* and *Type* apply to all the declarators in the declaration.

The declared type of a field is denoted by the *Type* that appears in the field declaration, followed by any bracket pairs that follow the *Identifier* in the declarator.

It is a compile-time error for the body of a class declaration to declare two fields with the same name.

If the class declares a field with a certain name, then the declaration of that field is said to *hide* any and all accessible declarations of fields with the same name in superclasses, and superinterfaces of the class.

If a field declaration hides the declaration of another field, the two fields need not have the same type.

A class inherits from its direct superclass and direct superinterfaces all the non-private fields of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.

Note that a *private* field of a superclass might be accessible to a subclass (for example, if both classes are members of the same class). Nevertheless, a *private* field is never inherited by a subclass.

It is possible for a class to inherit more than one field with the same name. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the class to refer to any such field by its simple name will result in a compile-time error, because such a reference is ambiguous.

There might be several paths by which the same field declaration might be inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

A value stored in a field of type `float` is always an element of the float value set (§4.2.3); similarly, a value stored in a field of type `double` is always an element of the double value set. It is not permitted for a field of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a field of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

8.3.1 Field Modifiers*FieldModifiers:**FieldModifier**FieldModifiers FieldModifier**FieldModifier: one of**Annotation* `public` `protected` `private``static` `final` `transient` `volatile`

If an annotation *a* (§9.7) on a field declaration corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.FIELD`, or a compile-time error occurs.

It is a compile-time error if the same modifier appears more than once in a field declaration, or if a field declaration has more than one of the access modifiers `public`, `protected`, and `private`.

8.3.1.1 *static Fields*

If a field is declared `static`, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a class variable, is incarnated when the class is initialized (§12.4).

A field that is not declared `static` (sometimes called a non-`static` field) is called an *instance variable*. Whenever a new instance of a class is created (§12.5), a new variable associated with that instance is created for every instance variable declared in that class or any of its superclasses.

8.3.1.2 *final Fields*

A field can be declared `final` (§4.12.4). Both class and instance variables (`static` and non-`static` fields) may be declared `final`.

It is a compile-time error if a blank `final` (§4.12.4) class variable is not definitely assigned (§16.8) by a static initializer (§8.7) of the class in which it is declared.

A blank `final` instance variable must be definitely assigned (§16.9) at the end of every constructor (§8.8) of the class in which it is declared; otherwise a compile-time error occurs.

8.3.1.3 *transient Fields*

Variables may be marked `transient` to indicate that they are not part of the persistent state of an object.

8.3.1.4 *volatile Fields*

A field may be declared `volatile`, in which case the Java Memory Model (§17.4) ensures that all threads see a consistent value for the variable.

It is a compile-time error if a `final` variable is also declared `volatile`.

8.3.2 Initialization of Fields

If a field declarator contains a *variable initializer*, then it has the semantics of an assignment (§15.26) to the declared variable, and:

- If the declarator is for a class variable (that is, a `static` field), then the variable initializer is evaluated and the assignment performed exactly once, when the class is initialized (§12.4).
- If the declarator is for an instance variable (that is, a field that is not `static`), then the variable initializer is evaluated and the assignment performed each time an instance of the class is created (§12.5).

8.3.2.1 *Initializers for Class Variables*

If a reference by simple name to any instance variable occurs in an initialization expression for a class variable, then a compile-time error occurs.

If the keyword `this` (§15.8.3) or the keyword `super` (§15.11.2, §15.12) occurs in an initialization expression for a class variable, then a compile-time error occurs.

8.3.2.2 *Initializers for Instance Variables*

Initialization expressions for instance variables may use the simple name of any static variable declared in or inherited by the class, even one whose declaration occurs textually later.

Initialization expressions for instance variables are permitted to refer to the current object `this` (§15.8.3) and to use the keyword `super` (§15.11.2, §15.12).

8.3.2.3 Restrictions on the use of Fields during Initialization

The declaration of a member needs to appear textually before it is used only if the member is an instance (respectively `static`) field of a class or interface *C* and all of the following conditions hold:

- The usage occurs in an instance (respectively `static`) variable initializer of *C* or in an instance (respectively `static`) initializer of *C*.
- The usage is not on the left hand side of an assignment.
- The usage is via a simple name.
- *C* is the innermost class or interface enclosing the usage.

It is a compile-time error if any of the four requirements above are not met.

These restrictions are designed to catch, at compile time, circular or otherwise malformed initializations.

8.4 Method Declarations

A *method* declares executable code that can be invoked, passing a fixed number of values as arguments.

MethodDeclaration:

MethodHeader *MethodBody*

MethodHeader:

*MethodModifiers*_{opt} *TypeParameters*_{opt} *Result* *MethodDeclarator* *Throws*_{opt}

Result:

Type

`void`

MethodDeclarator:

Identifier (*FormalParameterList*_{opt})

The *Identifier* in a *MethodDeclarator* may be used in a name to refer to the method.

The *Result* of a method declaration either declares the type of value that the method returns, or uses the keyword `void` to indicate that the method does not return a value.

For compatibility with older versions of the Java SE platform, the declaration of a method that returns an array is allowed to place (some or all of) the empty bracket pairs that form the declaration of the array type after the parameter list. This is supported by the obsolescent production:

MethodDeclarator:

MethodDeclarator []

but should not be used in new code.

It is a compile-time error for the body of a class to declare as members two methods with override-equivalent signatures (§8.4.2).

8.4.1 Formal Parameters

The *formal parameters* of a method or constructor, if any, are specified by a list of comma-separated parameter specifiers. Each parameter specifier consists of a type (optionally preceded by the `final` modifier and/or one or more annotations (§9.7)) and an identifier (optionally followed by brackets) that specifies the name of the parameter.

The last formal parameter in a list is special: it may be a *variable arity parameter*, indicated by an ellipsis following the type.

FormalParameterList:

LastFormalParameter

FormalParameters , *LastFormalParameter*

FormalParameters:

FormalParameter

FormalParameters , *FormalParameter*

FormalParameter:

*VariableModifiers*_{opt} *Type* *VariableDeclaratorId*

VariableModifiers:

VariableModifier

VariableModifiers *VariableModifier*

VariableModifier: one of

Annotation *final*

LastFormalParameter:

*VariableModifiers*_{opt} *Type* . . . *VariableDeclaratorId*

FormalParameter

If a method or constructor has no formal parameters, only an empty pair of parentheses appears in the declaration of the method or constructor.

If an annotation *a* (§9.7) on a formal parameter corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to *annotation.Target*, then *m* must have an element whose value is *annotation.ElementType.PARAMETER*, or a compile-time error occurs.

The declared type of a formal parameter is denoted by the *Type* that appears in its parameter specifier, followed by any bracket pairs that follow the *Identifier* in the declarator, except for a variable arity parameter, whose declared type is the *Type* that appears in its parameter specifier.

It is a compile-time error to use mixed array notation (§10.2) for a variable arity parameter.

It is a compile-time error if two formal parameters of the same method or constructor are declared to have the same name (that is, their declarations mention the same *Identifier*).

A formal parameter can only be referred to using a simple name (§6.5.6.1), not a qualified name.

It is a compile-time error if the name of a formal parameter is redeclared as a local variable of the method or constructor, or as an exception parameter of a catch clause in a try statement in the body of the method or constructor.

It is a compile-time error if a formal parameter that is declared `final` is assigned to within the body of the method or constructor.

When the method or constructor is invoked (§15.12), the values of the actual argument expressions initialize newly created parameter variables, each of the declared *Type*, before execution of the body of the method or constructor. The *Identifier* that appears in the *DeclaratorId* may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

If the last formal parameter is a variable arity parameter of type *T*, it is considered to define a formal parameter of type *T*[*]*. The method is then a *variable arity method*. Otherwise, it is a *fixed arity method*.

Invocations of a variable arity method may contain more actual argument expressions than formal parameters. All the actual argument expressions that do not correspond to the formal parameters preceding the variable arity parameter will be evaluated and the results stored into an array that will be passed to the method invocation (§15.12.4.2).

A method or constructor parameter of type `float` always contains an element of the float value set (§4.2.3); similarly, a method or constructor parameter of type `double` always contains an element of the double value set. It is not permitted for a method or constructor parameter of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a method parameter of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

Where an actual argument expression corresponding to a parameter variable is not FP-strict (§15.4), evaluation of that actual argument expression is permitted to use intermediate values drawn from the appropriate extended-exponent value sets. Prior to being stored in the parameter variable, the result of such an expression is mapped to the nearest value in the corresponding standard value set by method invocation conversion (§5.3).

8.4.2 Method Signature

It is a compile-time error to declare two methods with override-equivalent signatures in a class.

Two methods have the *same signature* if they have the same name and argument types.

Two method or constructor declarations M and N have the *same argument types* if all of the following conditions hold:

- They have the same number of formal parameters (possibly zero)
- They have the same number of type parameters (possibly zero)
- Let $A_1, \dots, A_n$ be the type parameters of M and let $B_1, \dots, B_n$ be the type parameters of N . After renaming each occurrence of a B_i in N 's type to A_i , the bounds of corresponding type variables are the same, and the formal parameter types of M and N are the same.

The signature of a method m_1 is a *subsignature* of the signature of a method m_2 if either:

- m_2 has the same signature as m_1 , or
- the signature of m_1 is the same as the erasure of the signature of m_2 .

Two method signatures m_1 and m_2 are *override-equivalent* iff either m_1 is a subsignature of m_2 or m_2 is a subsignature of m_1 .

8.4.3 Method Modifiers

MethodModifiers:

MethodModifier

MethodModifiers MethodModifier

MethodModifier: one of

Annotation public protected private abstract

static final synchronized native strictfp

If an annotation a (§9.7) on a method declaration corresponds to an annotation type T (§9.6), and T has a (meta-)annotation m that corresponds to `annotation.Target`, then m must have an element whose value is `annotation.ElementType.METHOD`, or a compile-time error occurs.

It is a compile-time error if the same modifier appears more than once in a method declaration, or if a method declaration has more than one of the access modifiers `public`, `protected`, and `private`.

It is a compile-time error if a method declaration that contains the keyword `abstract` also contains any one of the keywords `private`, `static`, `final`, `native`, `strictfp`, or `synchronized`.

It is a compile-time error if a method declaration that contains the keyword `native` also contains `strictfp`.

8.4.3.1 `abstract` *Methods*

An `abstract` method declaration introduces the method as a member, providing its signature (§8.4.2), return type, and throws clause (if any), but does not provide an implementation.

The declaration of an `abstract` method *m* must appear directly within an `abstract` class (call it *A*) unless it occurs within an `enum` (§8.9); otherwise a compile-time error occurs.

Every subclass of *A* that is not `abstract` (§8.1.1.1) must provide an implementation for *m*, or a compile-time error occurs.

An `abstract` class can override an `abstract` method by providing another `abstract` method declaration.

An instance method that is not `abstract` can be overridden by an `abstract` method.

8.4.3.2 `static` *Methods*

A method that is declared `static` is called a *class method*.

It is a compile-time error to use the name of a type parameter of any surrounding declaration in the header or body of a class method.

A class method is always invoked without reference to a particular object. It is a compile-time error to attempt to reference the current object using the keyword `this` or the keyword `super`.

A method that is not declared `static` is called an *instance method*, and sometimes called a *non-static method*.

An instance method is always invoked with respect to an object, which becomes the current object to which the keywords `this` and `super` refer during execution of the method body.

8.4.3.3 `final` *Methods*

A method can be declared `final` to prevent subclasses from overriding or hiding it.

It is a compile-time error to attempt to override or hide a `final` method.

A `private` method and all methods declared immediately within a `final` class (§8.1.1.2) behave as if they are `final`, since it is impossible to override them.

8.4.3.4 *native Methods*

A method that is `native` is implemented in platform-dependent code, typically written in another programming language such as C, C++, FORTRAN, or assembly language. The body of a `native` method is given as a semicolon only, indicating that the implementation is omitted, instead of a block.

8.4.3.5 *strictfp Methods*

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the method body be explicitly FP-strict (§15.4).

8.4.3.6 *synchronized Methods*

A `synchronized` method acquires a monitor (§17.1) before it executes.

For a class (`static`) method, the monitor associated with the `Class` object for the method's class is used. For an instance method, the monitor associated with `this` (the object for which the method was invoked) is used.

These are the same monitors that can be used by the `synchronized` statement (§14.19).

8.4.4 **Generic Methods**

A method is *generic* if it declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the method. The form of the type parameter section of a generic method is identical to the type parameter section of a generic class (§8.1.2).

8.4.5 **Method Return Type**

The *return type* of a method declares the type of value a method returns, if it returns a value, or states that the method is `void`.

A method declaration d_1 with return type R_1 is *return-type-substitutable* for another method d_2 with return type R_2 , if and only if the following conditions hold:

- If R_1 is `void` then R_2 is `void`.

- If R_1 is a primitive type, then R_2 is identical to R_1 .
- If R_1 is a reference type then:
 - ♦ R_1 is either a subtype of R_2 or R_1 can be converted to a subtype of R_2 by unchecked conversion (§5.1.9), or
 - ♦ $R_1 = |R_2|$

8.4.6 Method Throws

A **throws clause** is used to declare any checked exception classes (§11.1.1) that the statements in a method or constructor body can throw.

Throws:

throws ExceptionTypeList

ExceptionTypeList:

ExceptionType

ExceptionTypeList , ExceptionType

ExceptionType:

ClassType

TypeVariable

It is a compile-time error if any *ExceptionType* mentioned in a throws clause is not a subtype (§4.10) of *Throwable*.

It is permitted but not required to mention unchecked exception classes (§11.1.1) in a throws clause.

A method that overrides or hides another method (§8.4.8), including methods that implement abstract methods defined in interfaces, may not be declared to throw more checked exceptions than the overridden or hidden method.

More precisely, suppose that B is a class or interface, and A is a superclass or superinterface of B , and a method declaration n in B overrides or hides a method declaration m in A . If n has a throws clause that mentions any checked exception types, then m must have a throws clause, and for every checked exception type listed in the throws clause of n , that same exception class or one of its supertypes must occur in the erasure of the throws clause of m ; otherwise, a compile-time error occurs.

If the unerased throws clause of m does not contain a supertype of each exception type in the throws clause of n , an unchecked warning must be issued.

Type variables are allowed in a `throws` clause even though they are not allowed in a `catch` clause.

8.4.7 Method Body

A *method body* is either a block of code that implements the method or simply a semicolon, indicating the lack of an implementation.

The body of a method must be a semicolon if and only if the method is either *abstract* (§8.4.3.1) or *native* (§8.4.3.4).

```
MethodBody:  
    Block  
    ;
```

It is a compile-time error if a method declaration is either *abstract* or *native* and has a block for its body.

It is a compile-time error if a method declaration is neither *abstract* nor *native* and has a semicolon for its body.

If a method is declared *void*, then its body must not contain any *return* statement (§14.17) that has an *Expression*.

If a method is declared to have a return type, then every *return* statement (§14.17) in its body must have an *Expression*.

If a method is declared to have a return type, then a compile-time error occurs if the body of the method can complete normally (§14.1).

8.4.8 Inheritance, Overriding, and Hiding

A class *C* *inherits* from its direct superclass and direct superinterfaces all non-private methods (whether *abstract* or not) of the superclass and superinterfaces that are *public*, *protected*, or declared with default access in the same package as *C* and are neither overridden (§8.4.8.1) nor hidden (§8.4.8.2) by a declaration in the class.

If the method not inherited is declared in a class, or the method not inherited is declared in an interface and the new declaration is *abstract*, then the new declaration is said to *override* it.

If the method not inherited is *abstract* and the new declaration is not *abstract*, then the new declaration is said to *implement* it.

8.4.8.1 *Overriding (by Instance Methods)*

An instance method m_1 declared in a class C overrides another instance method, m_2 , declared in class A iff all of the following are true:

- C is a subclass of A .
- The signature of m_1 is a subsignature (§8.4.2) of the signature of m_2 .
- Either:
 - ♦ m_2 is `public`, `protected`, or declared with default access in the same package as C , or
 - ♦ m_1 overrides a method m_3 , m_3 distinct from m_1 , m_3 distinct from m_2 , such that m_3 overrides m_2 .

Moreover, if m_1 is not `abstract`, then m_1 is said to *implement* any and all declarations of `abstract` methods that it overrides.

It is a compile-time error if an instance method overrides a `static` method.

An overridden method can be accessed by using a method invocation expression (§15.12) that contains the keyword `super`. Note that a qualified name or a cast to a superclass type is not effective in attempting to access an overridden method; in this respect, overriding of methods differs from hiding of fields. See §15.12.4.4 for discussion and examples of this point.

The presence or absence of the `strictfp` modifier has absolutely no effect on the rules for overriding methods and implementing `abstract` methods. For example, it is permitted for a method that is not FP-strict to override an FP-strict method and it is permitted for an FP-strict method to override a method that is not FP-strict.

8.4.8.2 *Hiding (by Class Methods)*

If a class declares a `static` method m , then the declaration m is said to *hide* any method m' , where the signature of m is a subsignature (§8.4.2) of the signature of m' , in the superclasses and superinterfaces of the class that would otherwise be accessible to code in the class.

It is a compile-time error if a `static` method hides an instance method.

A hidden method can be accessed by using a qualified name or by using a method invocation expression (§15.12) that contains the keyword `super` or a cast to a superclass type. In this respect, hiding of methods is similar to hiding of fields.

8.4.8.3 *Requirements in Overriding and Hiding*

If a method declaration d_1 with return type R_1 overrides or hides the declaration of another method d_2 with return type R_2 , then d_1 must be return-type-substitutable (§8.4.5) for d_2 , or a compile-time error occurs.

Furthermore, if R_1 is not a subtype of R_2 , an unchecked warning must be issued (unless suppressed (§9.6.3.5)).

A method declaration must not have a `throws` clause that conflicts (§8.4.6) with that of any method that it overrides or hides; otherwise, a compile-time error occurs.

It is a compile-time error if a type declaration T has a member method m_1 and there exists a method m_2 declared in T or a supertype of T such that all of the following conditions hold:

- m_1 and m_2 have the same name.
- m_2 is accessible from T .
- The signature of m_1 is not a subsignature (§8.4.2) of the signature of m_2 .
- The signature of m_1 or some method m_1 overrides (directly or indirectly) has the same erasure as the signature of m_2 or some method m_2 overrides (directly or indirectly).

The access modifier (§6.6) of an overriding or hiding method must provide at least as much access as the overridden or hidden method, or a compile-time error occurs.

- If the overridden or hidden method is `public`, then the overriding or hiding method must be `public`; otherwise, a compile-time error occurs.
- If the overridden or hidden method is `protected`, then the overriding or hiding method must be `protected` or `public`; otherwise, a compile-time error occurs.
- If the overridden or hidden method has default (package) access, then the overriding or hiding method must *not* be `private`; otherwise, a compile-time error occurs.

8.4.8.4 *Inheriting Methods with Override-Equivalent Signatures*

It is possible for a class to inherit multiple methods with override-equivalent (§8.4.2) signatures.

It is a compile-time error if a class C inherits a concrete method whose signature is a subsignature of another concrete method inherited by C .

Otherwise, there are two possible cases:

- If one of the inherited methods is not abstract, then there are two subcases:
 - ♦ If the method that is not abstract is `static`, a compile-time error occurs.
 - ♦ Otherwise, the method that is not abstract is considered to override, and therefore to implement, all the other methods on behalf of the class that inherits it.

If the signature of the non-abstract method is not a subsignature of each of the other inherited methods, an unchecked warning must be issued (unless suppressed (§9.6.3.5)).

If the return type of the non-abstract method is not a subtype of the return type of any of the other inherited methods, an unchecked warning must be issued.

A compile-time error occurs if the return type of the non-abstract method is not return-type-substitutable (§8.4.5) for each of the other inherited methods.

A compile-time error occurs if the inherited method that is not abstract has a `throws` clause that conflicts (§8.4.6) with that of any other of the inherited methods.

- If all the inherited methods are abstract, then the class is necessarily an abstract class and is considered to inherit all the abstract methods.

One of the inherited methods must be return-type-substitutable for any other inherited method; otherwise, a compile-time error occurs. (The `throws` clauses do not cause errors in this case.)

There might be several paths by which the same method declaration might be inherited from an interface. This fact causes no difficulty and never, of itself, results in a compile-time error.

8.4.9 Overloading

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not override-equivalent, then the method name is said to be *overloaded*.

This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the `throws` clauses of two methods with the same name, unless their signatures are override-equivalent.

Methods are overridden on a signature-by-signature basis.

When a method is invoked (§15.12), the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked (§15.12.2). If the method that is to be invoked is an instance method, the actual method to be invoked will be determined at run time, using dynamic method lookup (§15.12.4).

8.5 Member Type Declarations

A *member class* is a class whose declaration is directly enclosed in another class or interface declaration.

A *member interface* is an interface whose declaration is directly enclosed in another class or interface declaration.

If the class declares a member type with a certain name, then the declaration of that type is said to *hide* any and all accessible declarations of member types with the same name in superclasses and superinterfaces of the class.

A class inherits from its direct superclass and direct superinterfaces all the non-private member types of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.

A class may inherit two or more type declarations with the same name, either from two interfaces or from its superclass and an interface. It is a compile-time error to attempt to refer to any ambiguously inherited class or interface by its simple name.

If the same type declaration is inherited from an interface by multiple paths, the class or interface is considered to be inherited only once. It may be referred to by its simple name without ambiguity.

8.5.1 Access Modifiers

It is a compile-time error if a member type declaration has more than one of the access modifiers `public`, `protected`, and `private` (§6.6).

A member interface in a class declaration is implicitly `public` unless an access modifier is specified.

Member type declarations may have annotation modifiers (§9.7) like any other type or member declaration.

8.5.2 Static Member Type Declarations

The `static` keyword may modify the declaration of a member type *C* within the body of a non-inner class or interface *T*. Its effect is to declare that *C* is not an inner class. Just as a `static` method of *T* has no current instance of *T* in its body, *C* also has no current instance of *T*, nor does it have any lexically enclosing instances.

It is a compile-time error if a `static` class contains a usage of a non-`static` member of an enclosing class.

Member interfaces are always implicitly `static`.

It is permitted but not required for the declaration of a member interface to explicitly list the `static` modifier.

8.6 Instance Initializers

An *instance initializer* declared in a class is executed when an instance of the class is created (§15.9), §8.8.7.1).

InstanceInitializer:
Block

It is a compile-time error if an instance initializer cannot complete normally (§14.21).

It is a compile-time error if a `return` statement (§14.17) appears anywhere within an instance initializer.

Instance initializers are permitted to refer to the current object via the keyword `this` (§15.8.3), to use the keyword `super` (§15.11.2, §15.12), and to use any type variables in scope.

8.7 Static Initializers

Any *static initializers* declared in a class are executed when the class is initialized (§12.4). Together with any field initializers for class variables (§8.3.2), static initializers may be used to initialize the class variables of the class.

StaticInitializer:
`static` *Block*

It is a compile-time error if a static initializer cannot complete normally (§14.21).

It is a compile-time error if a `return` statement (§14.17) appears anywhere within a static initializer.

It is a compile-time error if the keyword `this` (§15.8.3) or the keyword `super` (§15.11, §15.12) or any type variable declared outside the static initializer, appears anywhere within a static initializer.

The static initializers and class variable initializers of a class are executed in textual order (§12.4.2).

8.8 Constructor Declarations

A *constructor* is used in the creation of an object that is an instance of a class.

ConstructorDeclaration:

*ConstructorModifiers*_{opt} *ConstructorDeclarator*
*Throws*_{opt} *ConstructorBody*

ConstructorDeclarator:

*TypeParameters*_{opt} *SimpleTypeName* (*FormalParameterList*_{opt})

The *SimpleTypeName* in the *ConstructorDeclarator* must be the simple name of the class that contains the constructor declaration; otherwise a compile-time error occurs.

In all other respects, the constructor declaration looks just like a method declaration that has no result type.

Constructors are invoked by class instance creation expressions (§15.9), by the conversions and concatenations caused by the string concatenation operator `+` (§15.18.1), and by explicit constructor invocations from other constructors (§8.8.7).

Constructors are never invoked by method invocation expressions (§15.12).

Access to constructors is governed by access modifiers (§6.6).

Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.

8.8.1 Formal Parameters and Type Parameters

The formal parameters and type parameters of a constructor are identical in syntax and semantics to those of a method (§8.4.1).

8.8.2 Constructor Signature

It is a compile-time error to declare two constructors with override-equivalent (§8.4.2) signatures in a class.

It is a compile-time error to declare two constructors whose signatures have the same erasure (§4.6) in a class.

8.8.3 Constructor Modifiers

ConstructorModifiers:

ConstructorModifier

ConstructorModifiers ConstructorModifier

ConstructorModifier: one of

Annotation `public` `protected` `private`

If an annotation *a* (§9.7) on a constructor corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.CONSTRUCTOR`, or a compile-time error occurs.

It is a compile-time error if the same modifier appears more than once in a constructor declaration, or if a constructor declaration has more than one of the access modifiers `public`, `protected`, and `private`.

It is a compile-time error if the constructor of an enum type (§8.9) is declared `public` or `protected`.

If no access modifier is specified for the constructor of a normal class, the constructor has default access.

If no access modifier is specified for the constructor of an enum type, the constructor is `private`.

8.8.4 Generic Constructors

It is possible for a constructor to be declared generic, independently of whether the class the constructor is declared in is itself generic.

A constructor is *generic* if it declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the constructor. The form of the type parameter section of a generic constructor is identical to the type parameter section of a generic class (§8.1.2).

8.8.5 Constructor Throws

The throws clause for a constructor is identical in structure and behavior to the throws clause for a method (§8.4.6).

8.8.6 The Type of a Constructor

The type of a constructor consists of its signature and the exception types given its throws clause.

8.8.7 Constructor Body

The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass (§8.8.7.1).

ConstructorBody:

{ ExplicitConstructorInvocation_{opt} BlockStatements_{opt} }

It is a compile-time error for a constructor to directly or indirectly invoke itself through a series of one or more explicit constructor invocations involving `this`.

If the constructor is a constructor for an enum type (§8.9), it is a compile-time error for it to invoke the superclass constructor explicitly.

If a constructor body does not begin with an explicit constructor invocation and the constructor being declared is not part of the primordial class `Object`, then the constructor body is implicitly assumed by the Java compiler to begin with a superclass constructor invocation `"super();"` , an invocation of the constructor of its direct superclass that takes no arguments.

Except for the possibility of explicit constructor invocations, the body of a constructor is like the body of a method (§8.4.7).

A return statement (§14.17) may be used in the body of a constructor if it does not include an expression.

8.8.7.1 Explicit Constructor Invocations

ExplicitConstructorInvocation:

```
NonWildTypeArgumentsopt this ( ArgumentListopt ) ;
NonWildTypeArgumentsopt super ( ArgumentListopt ) ;
Primary . NonWildTypeArgumentsopt super ( ArgumentListopt ) ;
```

NonWildTypeArguments:

< ReferenceTypeList >

ReferenceTypeList:

ReferenceType

ReferenceTypeList , ReferenceType

Explicit constructor invocation statements can be divided into two kinds:

- *Alternate constructor invocations* begin with the keyword *this* (possibly prefaced with explicit type arguments). They are used to invoke an alternate constructor of the same class.
- *Superclass constructor invocations* begin with either the keyword *super* (possibly prefaced with explicit type arguments) or a *Primary* expression. They are used to invoke a constructor of the direct superclass.

Superclass constructor invocations may be subdivided:

- ♦ *Unqualified superclass constructor invocations* begin with the keyword *super* (possibly prefaced with explicit type arguments).
- ♦ *Qualified superclass constructor invocations* begin with a *Primary* expression.

They allow a subclass constructor to explicitly specify the newly created object's immediately enclosing instance with respect to the direct superclass (§8.1.3). This may be necessary when the superclass is an inner class.

An explicit constructor invocation statement in a constructor body may not refer to any instance variables or instance methods or inner classes declared in this class or any superclass, or use *this* or *super* in any expression; otherwise, a compile-time error occurs.

Let *C* be the class being instantiated, and let *S* be the direct superclass of *C*.

It is a compile-time error if *S* is not accessible (§6.6).

If a superclass constructor invocation statement is qualified, then:

- If S is not an inner class, or if the declaration of S occurs in a static context, then a compile-time error occurs.
- Otherwise, let p be the *Primary* expression immediately preceding ".super". Let O be the innermost lexically enclosing class of S . It is a compile-time error if the type of p is not O or a subclass of O , or if the type of p is not accessible (§6.6).

If a superclass constructor invocation statement is unqualified, and if S is an inner member class, then it is a compile-time error if S is not a member of a lexically enclosing class of C by declaration or inheritance.

Evaluation of an alternate constructor invocation statement proceeds by first evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.

Evaluation of a superclass constructor invocation statement is more complicated. Let C be the class being instantiated, let S be the direct superclass of C , and let i be the instance being created. The immediately enclosing instance of i with respect to S (if any) must be determined, as follows:

- If S is not an inner class, or if the declaration of S occurs in a static context, no immediately enclosing instance of i with respect to S exists.
- If the superclass constructor invocation is qualified, then the *Primary* expression p immediately preceding ".super" is evaluated.

If p evaluates to `null`, a `NullPointerException` is raised, and the superclass constructor invocation completes abruptly.

Otherwise, the result of this evaluation is the immediately enclosing instance of i with respect to S .

- If the superclass constructor invocation is not qualified, then:
 - ♦ If S is a local class (§14.3), then let O be the innermost lexically enclosing class of S . Let n be an integer such that O is the n 'th lexically enclosing class of C .
The immediately enclosing instance of i with respect to S is the n 'th lexically enclosing instance of `this`.
 - ♦ Otherwise, S is an inner member class (§8.5).

Let O be the innermost lexically enclosing class of S , and let n be an integer such that O is the n 'th lexically enclosing class of C .

The immediately enclosing instance of i with respect to S is the n 'th lexically enclosing instance of `this`.

After determining the immediately enclosing instance of *i* with respect to *S* (if any), evaluation of the superclass constructor invocation statement proceeds by evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.

Finally, if the superclass constructor invocation statement completes normally, then all instance variable initializers of *C* and all instance initializers of *C* are executed. If an instance initializer or instance variable initializer *I* textually precedes another instance initializer or instance variable initializer *J*, then *I* is executed before *J*.

Execution of instance variable initializers and instance initializers is performed regardless of whether the superclass constructor invocation actually appears as an explicit constructor invocation statement or is provided automatically. An alternate constructor invocation does not perform this additional implicit execution.

8.8.8 Constructor Overloading

Overloading of constructors is identical in behavior to overloading of methods. The overloading is resolved at compile time by each class instance creation expression (§15.9).

8.8.9 Default Constructor

If a class contains no constructor declarations, then a default constructor with no parameters is automatically provided.

If the class being declared is the primordial class `Object`, then the default constructor has an empty body. Otherwise, the default constructor simply invokes the superclass constructor with no arguments.

It is a compile-time error if a default constructor is provided by the Java compiler but the superclass does not have an accessible constructor (§6.6) that takes no arguments.

A default constructor has no throws clause.

In a class type, if the class is declared `public`, then the default constructor is implicitly given the access modifier `public` (§6.6); if the class is declared `protected`, then the default constructor is implicitly given the access modifier `protected` (§6.6); if the class is declared `private`, then the default constructor is implicitly given the access modifier `private` (§6.6); otherwise, the default constructor has the default access implied by no access modifier.

In an enum type, the default constructor is implicitly `private` (§8.9.2).

8.8.10 Preventing Instantiation of a Class

A class can be designed to prevent code outside the class declaration from creating instances of the class by declaring at least one constructor, to prevent the creation of an implicit constructor, and by declaring all constructors to be `private`.

A `public` class can likewise prevent the creation of instances outside its package by declaring at least one constructor, to prevent creation of a default constructor with public access, and by declaring no constructor that is `public`.

8.9 Enums

An *enum declaration* specifies a new enum type.

EnumDeclaration:

ClassModifiers_{opt} enum Identifier Interfaces_{opt} EnumBody

EnumBody:

{ EnumConstants_{opt} ,_{opt} EnumBodyDeclarations_{opt} }

Enum types (§8.9) must not be declared `abstract`; doing so will result in a compile-time error.

An enum type is implicitly `final` unless it contains at least one enum constant that has a class body.

It is a compile-time error to explicitly declare an enum type to be `final`.

Nested enum types are implicitly `static`. It is permissible to explicitly declare a nested enum type to be `static`.

The direct superclass of an enum type named *E* is `Enum<E>`.

An enum type has no instances other than those defined by its enum constants.

It is a compile-time error to attempt to explicitly instantiate an enum type (§15.9.1).

8.9.1 Enum Constants

The body of an enum type may contain *enum constants*. An enum constant defines an instance of the enum type.

EnumConstants:
EnumConstant
EnumConstants , *EnumConstant*

EnumConstant:
*Annotations*_{opt} *Identifier* *Arguments*_{opt} *ClassBody*_{opt}

Arguments:
 (*ArgumentList*_{opt})

EnumBodyDeclarations:
 ; *ClassBodyDeclarations*_{opt}

An enum constant may optionally be preceded by annotation modifiers. If an annotation *a* (§9.7) on an enum constant corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.FIELD`, or a compile-time error occurs.

The *Identifier* in a *EnumConstant* may be used in a name to refer to the enum constant.

An enum constant may be followed by arguments, which are passed to the constructor of the enum type when the constant is created during class initialization as described later in this section. The constructor to be invoked is chosen using the normal overloading rules (§15.12.2). If the arguments are omitted, an empty argument list is assumed.

The optional class body of an enum constant implicitly defines an anonymous class declaration (§15.9.5) that extends the immediately enclosing enum type. The class body is governed by the usual rules of anonymous classes; in particular it cannot contain any constructors.

It is a compile-time error for the class body of an enum constant to declare an abstract method.

Because there is only one instance of each enum constant, it is permissible to use the `==` operator in place of the `equals` method when comparing two object references if it is known that at least one of them refers to an enum constant.

8.9.2 Enum Body and Member Declarations

Any constructor or member declarations within an enum declaration apply to the enum type exactly as if they had been present in the class body of a normal class declaration, unless explicitly stated otherwise.

It is a compile-time error if a constructor declaration of an enum type is public or protected.

If an enum type has no constructor declarations, then a private constructor that takes no parameters (to match the implicit empty argument list) is automatically provided.

It is a compile-time error for an enum declaration to declare a finalizer.

An instance of an enum type may never be finalized.

It is a compile-time error for an enum type E to have an abstract method m as a member unless E has one or more enum constants, and all of E 's enum constants have class bodies that provide concrete implementations of m .

In addition to the members that an enum type E inherits from `Enum<E>`, for each declared enum constant with the name n , the enum type has an implicitly declared `public static final` field named n of type E . These fields are considered to be declared in the same order as the corresponding enum constants, before any `static` fields explicitly declared in the enum type. Each such field is initialized to the enum constant that corresponds to it. Each such field is also considered to be annotated by the same annotations as the corresponding enum constant. The enum constant is said to be *created* when the corresponding field is initialized.

In addition, if E is the name of an enum type, then that type has the following implicitly declared static methods:

```
/**
 * Returns an array containing the constants of this enum
 * type, in the order they're declared. This method may be
 * used to iterate over the constants as follows:
 *
 *     for(E c : E.values())
 *         System.out.println(c);
 *
 * @return an array containing the constants of this enum
 * type, in the order they're declared
 */
public static E[] values();

/**
 * Returns the enum constant of this type with the specified
 * name.
```

```
* The string must match exactly an identifier used to declare
* an enum constant in this type. (Extraneous whitespace
* characters are not permitted.)
*
* @return the enum constant with the specified name
* @throws IllegalArgumentException if this enum type has no
* constant with the specified name
*/
public static E valueOf(String name);
```

It is a compile-time error to reference a static field of an enum type that is not a compile-time constant (§15.28) from constructors, instance initializer blocks, or instance variable initializer expressions of that type.

It is a compile-time error for the constructors, instance initializer blocks, or instance variable initializer expressions of an enum constant *e* to refer to itself or to an enum constant of the same type that is declared to the right of *e*.

Interfaces

AN interface declaration introduces a new reference type whose members are classes, interfaces, constants, and abstract methods. This type has no implementation, but otherwise unrelated classes can implement it by providing implementations for its abstract methods.

A *nested interface* is any interface whose declaration occurs within the body of another class or interface.

A *top-level interface* is an interface that is not a nested interface.

We distinguish between two kinds of interfaces - normal interfaces and annotation types.

This chapter discusses the common semantics of all interfaces - normal interfaces, both top-level (§7.6) and nested (§8.5, §9.5), and annotation types (§9.6). Details that are specific to particular kinds of interfaces are discussed in the sections dedicated to these constructs.

Programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to `Object`.

An interface may be declared to be a *direct extension* of one or more other interfaces, meaning that it implicitly specifies all the member types, abstract methods, and constants of the interfaces it extends, except for any member types and constants that it may hide.

A class may be declared to *directly implement* one or more interfaces, meaning that any instance of the class implements all the abstract methods specified by the interface or interfaces. A class necessarily implements all the interfaces that its direct superclasses and direct superinterfaces do. This (multiple) interface inheritance allows objects to support (multiple) common behaviors without sharing any implementation.

A variable whose declared type is an interface type may have as its value a reference to any instance of a class which implements the specified interface. It is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclasses must actually be declared to implement the interface, or else the class is not considered to implement the interface.

9.1 Interface Declarations

An *interface declaration* specifies a new named reference type. There are two kinds of interface declarations - *normal interface declarations* and *annotation type declarations*.

InterfaceDeclaration:
NormalInterfaceDeclaration
AnnotationTypeDeclaration

Annotation types are described further in §9.6.

NormalInterfaceDeclaration:
*InterfaceModifiers*_{opt} *interface* *Identifier* *TypeParameters*_{opt}
*ExtendsInterfaces*_{opt} *InterfaceBody*

The *Identifier* in an interface declaration specifies the name of the interface.

It is a compile-time error if an interface has the same simple name as any of its enclosing classes or interfaces.

9.1.1 Interface Modifiers

An interface declaration may include *interface modifiers*.

InterfaceModifiers:
InterfaceModifier
InterfaceModifiers *InterfaceModifier*

InterfaceModifier: one of
Annotation *public* *protected* *private*
abstract *static* *strictfp*

If an annotation *a* (§9.7) on an interface declaration corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds

to `annotation.Target`, then `m` must have an element whose value is `annotation.ElementType.TYPE`, or a compile-time error occurs.

The access modifiers `protected` and `private` pertain only to member interfaces within a directly enclosing class or enum declaration (§8.5.1).

The modifier `static` pertains only to member interfaces (§8.5, §9.5), not to top level interfaces.

It is a compile-time error if the same modifier appears more than once in an interface declaration.

9.1.1.1 `abstract` Interfaces

Every interface is implicitly `abstract`.

9.1.1.2 `strictfp` Interfaces

The effect of the `strictfp` modifier is to make all `float` or `double` expressions within the interface declaration be explicitly FP-strict (§15.4).

This implies that all nested types declared in the interface are implicitly `strictfp`.

9.1.2 Generic Interfaces and Type Parameters

An interface is *generic* if it declares one or more type variables (§4.4).

These type variables are known as the *type parameters* of the interface. The type parameter section follows the interface name and is delimited by angle brackets.

In a interface's type parameter section, a type variable `T` *directly depends* on a type variable `S` if `S` is the bound of `T`, while `T` *depends* on `S` if either `T` directly depends on `S` or `T` directly depends on a type variable `U` that depends on `S` (using this definition recursively). It is a compile-time error if a type variable in a interface's type parameter section depends on itself.

A generic interface declaration defines a set of types, one for each possible invocation of the type parameter section. All parameterized types share the same interface at runtime.

It is a compile-time error to refer to a type parameter of an interface `I` anywhere in the declaration of a field or type member of `I`.

9.1.3 Superinterfaces and Subinterfaces

If an `extends` clause is provided, then the interface being declared extends each of the other named interfaces and therefore inherits the member types, methods, and constants of each of the other named interfaces.

These other named interfaces are the *direct superinterfaces* of the interface being declared.

Any class that implements the declared interface is also considered to implement all the interfaces that this interface extends.

ExtendsInterfaces:
`extends InterfaceTypeList`

Given a (possibly generic) interface declaration for $I\langle F_1, \dots, F_n \rangle$ ($n \geq 0$), the *direct superinterfaces* of the interface type (§4.5) $I\langle F_1, \dots, F_n \rangle$ are the types given in the `extends` clause of the declaration of I if an `extends` clause is present.

Let $I\langle F_1, \dots, F_n \rangle$ ($n > 0$), be a generic interface declaration. The direct superinterfaces of the parameterized interface type $I\langle T_1, \dots, T_n \rangle$, where T_i ($1 \leq i \leq n$) is a type, are all types $J\langle U_1 \ \theta, \dots, U_k \ \theta \rangle$, where $J\langle U_1, \dots, U_k \rangle$ is a direct superinterface of $I\langle F_1, \dots, F_n \rangle$, and θ is the substitution $[F_1 := T_1, \dots, F_n := T_n]$.

Each *InterfaceType* in the `extends` clause of an interface declaration must name an accessible (§6.6) interface type; otherwise a compile-time error occurs.

An interface I *directly depends* on a type T if T is mentioned in the `extends` clause of I either as a superinterface or as a qualifier within a superinterface name.

An interface I *depends* on a reference type T if any of the following conditions hold:

- I directly depends on T .
- I directly depends on a class C that depends (§8.1.5) on T .
- I directly depends on an interface J that depends on T (using this definition recursively).

It is a compile-time error if an interface depends on itself.

While every class is an extension of class `Object`, there is no single interface of which all interfaces are extensions.

The *superinterface* relationship is the transitive closure of the direct superinterface relationship. An interface K is a superinterface of interface I if either of the following is true:

- K is a direct superinterface of I .

- There exists an interface *J* such that *K* is a superinterface of *J*, and *J* is a superinterface of *I*, applying this definition recursively.

Interface *I* is said to be a *subinterface* of interface *K* whenever *K* is a superinterface of *I*.

9.1.4 Interface Body and Member Declarations

The body of an interface may declare members of the interface, that is, fields (§9.3), methods (§9.4), classes (§9.5), and interfaces (§9.5).

InterfaceBody:

{ *InterfaceMemberDeclarations*_{opt} }

InterfaceMemberDeclarations:

InterfaceMemberDeclaration

InterfaceMemberDeclarations *InterfaceMemberDeclaration*

InterfaceMemberDeclaration:

ConstantDeclaration

AbstractMethodDeclaration

ClassDeclaration

InterfaceDeclaration

;

9.2 Interface Members

The members of an interface are:

- Those members declared in the interface.
- Those members inherited from direct superinterfaces.
- If an interface has no direct superinterfaces, then the interface implicitly declares a `public abstract` member method *m* with signature *s*, return type *r*, and throws clause *t* corresponding to each `public` instance method *m* with signature *s*, return type *r*, and throws clause *t* declared in `Object`, unless a method with the same signature, same return type, and a compatible throws clause is explicitly declared by the interface.

It is a compile-time error if the interface explicitly declares such a method *m* in the case where *m* is declared to be `final` in `Object`.

It follows that is a compile-time error if the interface declares a method with a signature that is override-equivalent (§8.4.2) to a `public` method of `Object`, but has a different return type or incompatible throws clause.

The interface inherits, from the interfaces it extends, all members of those interfaces, except for fields, classes, and interfaces that it hides and methods that it overrides.

Fields, methods, and member types of an interface type may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures (§6.5). However, this is discouraged as a matter of style.

9.3 Field (Constant) Declarations

ConstantDeclaration:

*ConstantModifiers*_{opt} *Type* *VariableDeclarators* ;

ConstantModifiers:

ConstantModifier

ConstantModifier *ConstantModifiers*

ConstantModifier: one of

Annotation `public` `static` `final`

If an annotation *a* (§9.7) on a field declaration corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.FIELD`, or a compile-time error occurs.

Every field declaration in the body of an interface is implicitly `public`, `static`, and `final`. It is permitted to redundantly specify any or all of these modifiers for such fields.

It is a compile-time error if the same modifier appears more than once in a field declaration.

The declared type of a field is denoted by the *Type* that appears in the field declaration, followed by any bracket pairs that follow the *Identifier* in the declarator.

If the interface declares a field with a certain name, then the declaration of that field is said to *hide* any and all accessible declarations of fields with the same name in superinterfaces of the interface.

It is a compile-time error for the body of an interface declaration to declare two fields with the same name.

It is possible for an interface to inherit more than one field with the same name. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the interface to refer to any such field by its simple name will result in a compile-time error, because such a reference is ambiguous.

There might be several paths by which the same field declaration might be inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

9.3.1 Initialization of Fields in Interfaces

Every field in the body of an interface must have an initialization expression, which need not be a constant expression.

The variable initializer is evaluated and the assignment performed exactly once, when the interface is initialized (§12.4).

It is a compile-time error if an initialization expression for an interface field contains a reference by simple name to the same field or to another field whose declaration occurs textually later in the same interface.

One subtlety here is that, at run time, fields that are initialized with compile-time constant values are initialized first. This applies also to `static final` fields in classes (§8.3.2.1). This means, in particular, that these fields will never be observed to have their default initial values (§4.12.5), even by devious programs. See §12.4.2 and §13.4.9 for more discussion.

If the keyword `this` (§15.8.3) or the keyword `super` (§15.11.2, §15.12) occurs in an initialization expression for a field of an interface, then unless the occurrence is within the body of an anonymous class (§15.9.5), a compile-time error occurs.

9.4 Abstract Method Declarations

AbstractMethodDeclaration:

*AbstractMethodModifiers_{opt} TypeParameters_{opt} Result
MethodDeclarator Throws_{opt} ;*

AbstractMethodModifiers:

*AbstractMethodModifier
AbstractMethodModifiers AbstractMethodModifier*

AbstractMethodModifier: one of

Annotation public abstract

If an annotation *a* (§9.7) on a method declaration corresponds to an annotation type *T* (§9.6), and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.METHOD`, or a compile-time error occurs.

Every method declaration in the body of an interface is implicitly `public` (§6.6).

Every method declaration in the body of an interface is implicitly `abstract`, so its body is always represented by a semicolon, not a block.

It is permitted, but discouraged as a matter of style, to redundantly specify the `public` and/or `abstract` modifier for a method declared in an interface.

It is a compile-time error if the same modifier appears more than once on a method declared in an interface.

It is a compile-time error if a method declared in an interface is declared `static`, because `static` methods cannot be `abstract`.

It is a compile-time error if a method declared in an interface is `strictfp` or `native` or `synchronized`, because those keywords describe implementation properties rather than interface properties.

However, a method declared in an interface may be implemented by a method that is declared `strictfp` or `native` or `synchronized` in a class that implements the interface.

It is a compile-time error if a method declared in an interface is declared `final`.

However, a method declared in an interface may be implemented by a method that is declared `final` in a class that implements the interface.

It is a compile-time error for the body of an interface to declare, explicitly or implicitly, two methods with override-equivalent signatures (§8.4.2).

However, an interface may inherit several methods with such signatures (§9.4.1).

A method in an interface may be generic. The rules for type parameters of a generic method in an interface are the same as for a generic method in a class (§8.4.4).

9.4.1 Inheritance and Overriding

An instance method m_1 declared in an interface I overrides another instance method, m_2 , declared in interface J iff both of the following are true:

- I is a subinterface of J .
- The signature of m_1 is a subsignature (§8.4.2) of the signature of m_2 .

If a method declaration d_1 with return type R_1 overrides or hides the declaration of another method d_2 with return type R_2 , then d_1 must be return-type-substitutable (§8.4.5) for d_2 , or a compile-time error occurs.

Furthermore, if R_1 is not a subtype of R_2 , an unchecked warning must be issued.

Moreover, a method declaration must not have a `throws` clause that conflicts (§8.4.6) with that of any method that it overrides; otherwise, a compile-time error occurs.

It is a compile-time error if a type declaration T has a member method m_1 and there exists a method m_2 declared in T or a supertype of T such that all of the following conditions hold:

- m_1 and m_2 have the same name.
- m_2 is accessible from T .
- The signature of m_1 is not a subsignature (§8.4.2) of the signature of m_2 .
- The signature of m_1 or some method m_1 overrides (directly or indirectly) has the same erasure as the signature of m_2 or some method m_2 overrides (directly or indirectly).

Methods are overridden on a signature-by-signature basis. If, for example, an interface declares two `public` methods with the same name, and a subinterface overrides one of them, the subinterface still inherits the other method.

An interface inherits from its direct superinterfaces all methods of the superinterfaces that are not overridden by a declaration in the interface.

It is possible for an interface to inherit several methods with override-equivalent signatures (§8.4.2). Such a situation does not in itself cause a compile-time error. The interface is considered to inherit all the methods.

However, one of the inherited methods must be return-type-substitutable for any other inherited method; otherwise, a compile-time error occurs. (The `throws` clauses do not cause errors in this case.)

There might be several paths by which the same method declaration is inherited from an interface. This fact causes no difficulty and never, of itself, results in a compile-time error.

9.4.2 Overloading

If two methods of an interface (whether both declared in the same interface, or both inherited by an interface, or one declared and one inherited) have the same name but different signatures that are not override-equivalent (§8.4.2), then the method name is said to be *overloaded*.

This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the `throws` clauses of two methods with the same name but different signatures that are not override-equivalent.

9.5 Member Type Declarations

Interfaces may contain member type declarations (§8.5).

A member type declaration in an interface is implicitly `static` and `public`. It is permitted to redundantly specify either or both of these modifiers.

It is a compile-time error if the same modifier appears more than once in a member type declaration in an interface.

If a member type declared with simple name *C* is directly enclosed within the declaration of an interface with fully qualified name *N*, then the member type has the fully qualified name *N.C*.

If the interface declares a member type with a certain name, then the declaration of that type is said to *hide* any and all accessible declarations of member types with the same name in superinterfaces of the interface.

An interface inherits from its direct superinterfaces all the non-private member types of the superinterfaces that are both accessible to code in the interface and not hidden by a declaration in the interface.

An interface may inherit two or more type declarations with the same name. It is a compile-time error to attempt to refer to any ambiguously inherited class or interface by its simple name.

If the same type declaration is inherited from an interface by multiple paths, the class or interface is considered to be inherited only once; it may be referred to by its simple name without ambiguity.

9.6 Annotation Types

An *annotation type declaration* is a special kind of interface declaration. To distinguish an annotation type declaration from an ordinary interface declaration, the keyword `interface` is preceded by an at-sign (`@`).

AnnotationTypeDeclaration:

InterfaceModifiers_{opt} @ interface Identifier AnnotationTypeBody

AnnotationTypeBody:

{ AnnotationTypeElementDeclarations_{opt} }

AnnotationTypeElementDeclarations:

AnnotationTypeElementDeclaration

AnnotationTypeElementDeclarations AnnotationTypeElementDeclaration

If an annotation `a` on an annotation type declaration corresponds to an annotation type `T`, and `T` has a (meta-)annotation `m` that corresponds to `annotation.Target`, then `m` must have either an element whose value is `annotation.ElementType.ANNOTATION_TYPE`, or an element whose value is `annotation.ElementType.TYPE`, or a compile-time error occurs.

The *Identifier* in an annotation type declaration specifies the name of the annotation type.

It is a compile-time error if an annotation type has the same simple name as any of its enclosing classes or interfaces.

The direct superinterface of an annotation type is always `annotation.Annotation`.

An annotation type declaration inherits several members from `annotation.Annotation`, including the implicitly declared methods corresponding to the instance methods in `Object`, yet these methods do not define elements (§9.6.1) of the annotation type and it is illegal to use them in annotations.

Unless explicitly modified herein, all of the rules that apply to ordinary interface declarations apply to annotation type declarations.

9.6.1 Annotation Type Elements

Each method declaration in an annotation type declaration defines an *element* of the annotation type.

Annotation types can have zero or more elements. An annotation type has no elements other than those defined by the methods it explicitly declares.

AnnotationTypeElementDeclaration:

```
AbstractMethodModifiersopt Type Identifier (   ) Dimsopt DefaultValueopt ;  
ConstantDeclaration  
ClassDeclaration  
InterfaceDeclaration  
EnumDeclaration  
AnnotationTypeDeclaration  
;
```

DefaultValue:

```
default ElementValue
```

It is a compile-time error if the return type of a method declared in an annotation type is not one of the following: a primitive type, `String`, `Class`, any parameterized invocation of `Class`, an enum type (§8.9), an annotation type, or an array type (Chapter 10, *Arrays*) whose element type is one of the preceding types.

It is a compile-time error if any method declared in an annotation type has a signature that is override-equivalent to that of any `public` or `protected` method declared in class `Object` or in the interface `annotation.Annotation`.

It is a compile-time error if an annotation type declaration *T* contains an element of type *T*, either directly or indirectly.

By convention, the name of the sole element in a single-element annotation type is `value`. Linguistic support for this convention is provided by the single element annotation construct (§9.7.3); one must obey the convention in order to take advantage of the construct.

9.6.2 Defaults for Annotation Type Elements

An annotation type element may have a default value specified for it. This is done by following its (empty) parameter list with the keyword `default` and the default value of the element.

Defaults are applied dynamically at the time annotations are read; default values are not compiled into annotations. Thus, changing a default value affects annotations even in classes that were compiled before the change was made (presuming these annotations lack an explicit value for the defaulted element).

An *ElementValue* (§9.7) is used to specify a default value.

It is a compile-time error if the type of the element is not commensurate (§9.7) with the default value specified.

9.6.3 Predefined Annotation Types

Several annotation types are predefined in the libraries of the Java SE platform. Some of these predefined annotation types have special semantics. These semantics are specified in this section. This section does not provide a complete specification for the predefined annotations contained here in; that is the role of the appropriate API specifications. Only those semantics that require special behavior on the part of a Java compiler or Java virtual machine implementation are specified here.

9.6.3.1 *Target*

The annotation type `annotation.Target` is intended to be used in meta-annotations that indicate the kind of program element that an annotation type is applicable to.

`annotation.Target` has one element, of type `annotation.ElementType[]`.

It is a compile-time error if a given enum constant appears more than once in an annotation whose corresponding type is `annotation.Target`.

9.6.3.2 *Retention*

Annotations may be present only in source code, or they may be present in the binary form of a class or interface. An annotation that is present in the binary form may or may not be available at run-time via the reflection libraries of the Java SE platform. The annotation type `annotation.Retention` is used to choose among these possibilities.

If an annotation `a` corresponds to a type `T`, and `T` has a (meta-)annotation `m` that corresponds to `annotation.Retention`, then:

- If `m` has an element whose value is `annotation.RetentionPolicy.SOURCE`, then a Java compiler must ensure that `a` is not present in the binary representation of the class or interface in which `a` appears.
- If `m` has an element whose value is `annotation.RetentionPolicy.CLASS`, or `annotation.RetentionPolicy.RUNTIME`, then a Java compiler must ensure that `a` is represented in the binary representation of the class or interface in which `a` appears, unless `m` annotates a local variable declaration.

An annotation on a local variable declaration is never retained in the binary representation.

In addition, if `m` has an element whose value is `annotation.RetentionPolicy.RUNTIME`, the reflection libraries of the Java SE platform must make `a` available at run-time.

If `T` does not have a (meta-)annotation `m` that corresponds to `annotation.Retention`, then a Java compiler must treat `T` as if it does have such a meta-annotation `m` with an element whose value is `annotation.RetentionPolicy.CLASS`.

9.6.3.3 *Inherited*

The annotation type `annotation.Inherited` is used to indicate that annotations on a class `C` corresponding to a given annotation type are inherited by subclasses of `C`.

9.6.3.4 *Override*

Programmers occasionally overload a method declaration when they mean to override it, leading to subtle problems. The annotation type `Override` supports early detection of such problems.

If a method declaration is annotated with the annotation `@Override`, but the method does not in fact override or implement a method of a supertype, or a `public` method of `Object`, a compile-time error will occur.

9.6.3.5 *SuppressWarnings*

The annotation type `SuppressWarnings` supports programmer control over warnings otherwise issued by a Java compiler. It contains a single element that is an array of `String`.

If a program declaration is annotated with the annotation `@SuppressWarnings(value = { $S_1$ , ...,  $S_k$ })`, then a Java compiler must not report any warning identified by one of S_1 ... S_k if that warning would have been generated as a result of the annotated declaration or any of its parts.

Unchecked warnings are identified by the string "unchecked".

9.6.3.6 *Deprecated*

A program element annotated `@Deprecated` is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists.

A Java compiler must produce a warning when a deprecated type, method, field, or constructor is used (overridden, invoked, or referenced by name) unless:

- The use is within an entity that itself is annotated with the annotation `@Deprecated`; or
- The declaration and use are both within the same outermost class; or
- The use site is within an entity that is annotated to suppress the warning with the annotation `@SuppressWarnings("deprecation")`.

Use of the annotation `@Deprecated` on a local variable declaration or on a parameter declaration has no effect.

9.7 Annotations

An *annotation* is a modifier consisting of the name of an annotation type (§9.6) and zero or more element-value pairs, each of which associates a value with a different element of the annotation type.

The purpose of an annotation is simply to associate information with the annotated program element.

Annotations must contain an element-value pair for every element of the corresponding annotation type, except for those elements with default values, or a compile-time error occurs.

Annotations may, but are not required to, contain element-value pairs for elements with default values.

Annotations may be used as modifiers in any declaration, whether package (§7.4.1), class (§8.1.1) (including enums (§8.9)), interface (§9.1.1) (including annotation

types (§9.6)), field (§8.3.1, §9.3), method (§8.4.3, §9.4), formal parameter (§8.4.1), constructor (§8.8.3), or local variable (§14.4.1).

Annotations may also be used on enum constants. Such annotations are placed immediately before the enum constant they annotate.

It is a compile-time error if a declaration is annotated with more than one annotation for a given annotation type.

Annotations:

Annotation

Annotations Annotation

Annotation:

NormalAnnotation

MarkerAnnotation

SingleElementAnnotation

There are three kinds of annotations. The first (normal annotation) is fully general. The others (marker annotation and single-element annotation) are merely shorthands.

9.7.1 Normal Annotations

A *normal annotation* is used to annotate a program element.

NormalAnnotation:

@ TypeName (ElementValuePairs_{opt})

ElementValuePairs:

ElementValuePair

ElementValuePairs , ElementValuePair

ElementValuePair:

Identifier = ElementValue

ElementValue:

ConditionalExpression

Annotation

ElementValueArrayInitializer

ElementValueArrayInitializer:

{ ElementValues_{opt} , _{opt} }

ElementValues:

ElementValue

ElementValues , ElementValue

The *TypeName* names the annotation type corresponding to the annotation.

It is a compile-time error if *TypeName* does not name an annotation type that is accessible (§6.6) at the point where the annotation is used.

The *Identifier* in an *ElementValuePair* must be the simple name of one of the elements (i.e. methods) of the annotation type identified by *TypeName*; otherwise, a compile-time error occurs.

The return type of this method defines the element type of the element-value pair.

An element type *T* is *commensurate* with an element value *V* if and only if one of the following conditions is true:

- *T* is an array type *E[]* and either:
 - ♦ *V* is an *ElementValueArrayInitializer* and each *ElementValue* (analogous to a *VariableInitializer* in an array initializer) in *V* is commensurate with *E*; or
 - ♦ *V* is an *ElementValue* that is commensurate with *E*.
- The type of *V* is assignment compatible (§5.2) with *T*, and furthermore:
 - ♦ If *T* is a primitive type or `String`, and *V* is a constant expression (§15.28).

- ♦ *V* is not `null`.
- ♦ If *T* is `Class`, or an invocation of `Class`, and *V* is a class literal (§15.8.2).
- ♦ If *T* is an enum type, and *V* is an enum constant.

It is a compile-time error if the element type is not commensurate with the *ElementValue*.

If the element type is not an annotation type or an array type, *ElementValue* must be a *ConditionalExpression* (§15.25).

If the element type is an array type and the corresponding *ElementValue* is not an *ElementValueArrayInitializer*, then an array value whose sole element is the value represented by the *ElementValue* is associated with the element. Otherwise, if the corresponding *ElementValue* is an *ElementValueArrayInitializer*, then the array value represented by the *ElementValueArrayInitializer* is associated with the element.

An *ElementValue* is always FP-strict (§15.4).

An annotation on an annotation type declaration is known as a *meta-annotation*.

An annotation type may be used to annotate its own declaration. More generally, circularities in the transitive closure of the "annotates" relation are permitted.

9.7.2 Marker Annotations

The second form of annotation, *marker annotation*, is a shorthand designed for use with marker annotation types.

MarkerAnnotation:
@ Identifier

It is shorthand for the normal annotation:

`@Identifier()`

Note that it is legal to use the marker annotation form for annotation types with elements, so long as all the elements have default values.

9.7.3 Single-Element Annotations

The third form of annotation, *single-element annotation*, is a shorthand designed for use with single-element annotation types.

SingleElementAnnotation:

@ Identifier (ElementValue)

It is shorthand for the normal annotation:

`@Identifier(value = ElementValue)`

Note that it is legal to use single-element annotations for annotation types with multiple elements, so long as one element is named `value`, and all other elements have default values.

Arrays

IN the Java programming language, *arrays* are objects (§4.3.1), are dynamically created, and may be assigned to variables of type `Object` (§4.3.2). All methods of class `Object` may be invoked on an array.

An array object contains a number of variables. The number of variables may be zero, in which case the array is said to be *empty*. The variables contained in an array have no names; instead they are referenced by array access expressions that use non-negative integer index values. These variables are called the *components* of the array. If an array has n components, we say n is the length of the array; the components of the array are referenced using integer indices from 0 to $n - 1$, inclusive.

All the components of an array have the same type, called the *component type* of the array. If the component type of an array is T , then the type of the array itself is written $T[]$.

The value of an array component of type `float` is always an element of the float value set (§4.2.3); similarly, the value of an array component of type `double` is always an element of the double value set. It is not permitted for the value of an array component of type `float` to be an element of the float-extended-exponent value set that is not also an element of the float value set, nor for the value of an array component of type `double` to be an element of the double-extended-exponent value set that is not also an element of the double value set.

The component type of an array may itself be an array type. The components of such an array may contain references to subarrays. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; this is called the *element type* of the original array, and the components at this level of the data structure are called the *elements* of the original array.

There are some situations in which an element of an array can be an array: if the element type is `Object` or `Cloneable` or `java.io.Serializable`, then some or all of the elements may be arrays, because any array object can be assigned to any variable of these types.

10.1 Array Types

Array types are used in declarations and in cast expressions (§15.16).

An array type is written as the name of an element type followed by some number of empty pairs of square brackets `[]`. The number of bracket pairs indicates the depth of array nesting.

An array's length is not part of its type.

The element type of an array may be any type, whether primitive or reference. In particular:

- Arrays with an interface type as the element type are allowed. An element of such an array may have as its value a null reference or an instance of any type that implements the interface.
- Arrays with an abstract class type as the element type are allowed. An element of such an array may have as its value a null reference or an instance of any subclass of the abstract class that is not itself abstract.

The direct superclass of an array type is `Object`.

Every array type implements the interfaces `Cloneable` and `java.io.Serializable`.

10.2 Array Variables

A variable of array type holds a reference to an object. Declaring a variable of array type does not create an array object or allocate any space for array components. It creates only the variable itself, which can contain a reference to an array.

However, the initializer part of a declarator (§8.3, §9.3, §14.4.1) may create an array, a reference to which then becomes the initial value of the variable.

The `[]` may appear as part of the type at the beginning of the declaration, or as part of the declarator for a particular variable, or both.

In a variable declaration (§8.3, §8.4.1, §9.3, §14.14, §14.20) except for a variable arity parameter or `*`, the array type of a variable is denoted by the array type that appears at the beginning of the declaration, followed by any bracket pairs that follow the variable's *Identifier* in the declarator.

We do not recommend "mixed notation" in an array variable declaration, where brackets appear on both the type and in declarators.

Once an array object is created, its length never changes. To make an array variable refer to an array of different length, a reference to a different array must be assigned to the variable.

A single variable of array type may contain references to arrays of different lengths, because an array's length is not part of its type.

If an array variable *v* has type *A*[], where *A* is a reference type, then *v* can hold a reference to an instance of any array type *B*[], provided *B* can be assigned to *A*. This may result in a run-time exception on a *later* assignment; see §10.5 for a discussion.

10.3 Array Creation

An array is created by an array creation expression (§15.10) or an array initializer (§10.6).

An array creation expression specifies the element type, the number of levels of nested arrays, and the length of the array for at least one of the levels of nesting. The array's length is available as a `final` instance variable `length`.

It is a compile-time error if the element type is not a reifiable type (§4.7)

An array initializer creates an array and provides initial values for all its components.

10.4 Array Access

A component of an array is accessed by an array access expression (§15.13) that consists of an expression whose value is an array reference followed by an indexing expression enclosed by `[` and `]`, as in `A[i]`. All arrays are 0-origin. An array with length *n* can be indexed by the integers 0 to *n* - 1.

Arrays must be indexed by `int` values; `short`, `byte`, or `char` values may also be used as index values because they are subjected to unary numeric promotion (§5.6.1) and become `int` values.

An attempt to access an array component with a `long` index value results in a compile-time error.

All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown.

10.5 Array Store Exception

An assignment to an element of an array whose type is `A[]`, where `A` is a reference type, is checked at run-time to ensure that the value assigned can be assigned to the actual element type of the array, where the actual element type may be any reference type that is assignable to `A`.

If the value assigned to the element is not assignment-compatible (§5.2) with the actual element type, an `ArrayStoreException` is thrown.

10.6 Array Initializers

An *array initializer* may be specified in a declaration (§8.3, §9.3, §14.4), or as part of an array creation expression (§15.10), to create an array and provide some initial values.

ArrayInitializer:
 { *VariableInitializers*_{opt} ,_{opt} }

VariableInitializers:
 VariableInitializer
 VariableInitializers , *VariableInitializer*

An array initializer is written as a comma-separated list of expressions, enclosed by braces `{` and `}`.

A trailing comma may appear after the last expression in an array initializer and is ignored.

The length of the array to be constructed is equal to the number of variable initializers immediately enclosed by the braces of the array initializer. Space is allocated for a new array of that length. If there is insufficient space to allocate the array, evaluation of the array initializer completes abruptly by throwing an `OutOfMemoryError`. Otherwise, a one-dimensional array is created of the specified length, and each component of the array is initialized to its default value (§4.12.5).

The variable initializers immediately enclosed by the braces of the array initializer are then executed from left to right in the textual order they occur in the source code. The n 'th variable initializer specifies the value of the $n-1$ 'th array component. If execution of a variable initializer completes abruptly, then execution of the array initializer completes abruptly for the same reason. If all the variable initializer expressions complete normally, the array initializer completes normally, with the value of the newly initialized array.

Each variable initializer must be assignment-compatible (§5.2) with the array's component type, or a compile-time error occurs.

It is a compile-time error if the component type of the array being initialized is not reifiable (§4.7).

If the component type is an array type, then the variable initializer specifying a component may itself be an array initializer; that is, array initializers may be nested. In this case, execution of the nested array initializer constructs and initializes an array object by recursive application of this algorithm, and assigns it to the component.

10.7 Array Members

The members of an array type are all of the following:

- The `public final` field `length`, which contains the number of components of the array. `length` may be positive or zero.
- The `public` method `clone`, which overrides the method of the same name in class `Object` and throws no checked exceptions. The return type of the `clone` method of an array type `T[]` is `T[]`.
- All the members inherited from class `Object`; the only method of `Object` that is not inherited is its `clone` method.

A clone of a multidimensional array is shallow, which is to say that it creates only a single new array. Subarrays are shared.

10.8 **Class Objects for Arrays**

Every array has an associated `Class` object, shared with all other arrays with the same component type.

10.9 **An Array of Characters is Not a `String`**

In the Java programming language, unlike C, an array of `char` is not a `String`, and neither a `String` nor an array of `char` is terminated by `'\u0000'` (the NUL character).

A `String` object is immutable, that is, its contents never change, while an array of `char` has mutable elements. The method `toCharArray` in class `String` returns an array of characters containing the same character sequence as a `String`. The class `StringBuffer` implements useful methods on mutable arrays of characters.

Exceptions

WHEN a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an *exception*. An example of such a violation is an attempt to index outside the bounds of an array.

Some programming languages and their implementations react to such errors by peremptorily terminating the program; other programming languages allow an implementation to react in an arbitrary or unpredictable way. Neither of these approaches is compatible with the design goals of the Java SE platform: to provide portability and robustness.

Instead, the Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be *thrown* from the point where it occurred and is said to be *caught* at the point to which control is transferred.

Programs can also throw exceptions explicitly, using `throw` statements (§14.18).

Explicit use of `throw` statements provides an alternative to the old-fashioned style of handling error conditions by returning funny values, such as the integer value `-1` where a negative value would not normally be expected. Experience shows that too often such funny values are ignored or not checked for by callers, leading to programs that are not robust, exhibit undesirable behavior, or both.

Every exception is represented by an instance of the class `Throwable` or one of its subclasses; such an object can be used to carry information from the point at which an exception occurs to the handler that catches it. Handlers are established by `catch` clauses of `try` statements (§14.20). During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates

that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception. If no such handler is found, then the exception may be handled by the current thread's uncaught exception handler, or else by the uncaught exception handler of the `ThreadGroup` that is the parent of the current thread, or else by the global uncaught exception handler - thus every effort is made to avoid letting an exception go unhandled.

The exception mechanism of the Java SE platform is integrated with its synchronization model (§17.1), so that locks are released as synchronized statements (§14.19) and invocations of synchronized methods (§8.4.3.6, §15.12) complete abruptly.

This chapter describes the hierarchy of classes, rooted at `Throwable`, that represent exceptions, and gives an overview of the causes of exceptions (§11.1). It details how exceptions are checked at compile-time (§11.2) and processed at run-time (§11.3).

11.1 The Kinds and Causes of Exceptions

11.1.1 The Kinds of Exceptions

An exception is represented by an instance of the class `Throwable` (a direct subclass of `Object`) or one of its subclasses.

`Throwable` and all its subclasses are, collectively, the *exception classes*.

The classes `Exception` and `Error` are direct subclasses of `Throwable`.

`Exception` is the superclass of all the exceptions that ordinary programs may wish to recover from.

`Error` and all its subclasses are, collectively, the *error classes*. They are exceptions from which ordinary programs are not ordinarily expected to recover.

The class `RuntimeException` is a direct subclass of `Exception`. `RuntimeException` and all its subclasses are, collectively, the *runtime exception classes*. They are exceptions which may be thrown for many reasons during expression evaluation, but from which recovery may still be possible.

The *unchecked exception classes* are the runtime exception classes and the error classes.

The *checked exception classes* are all exception classes other than the unchecked exception classes. That is, the checked exception classes are all subclasses of `Exception` other than `RuntimeException` and its subclasses.

11.1.2 The Causes of Exceptions

An exception is thrown for one of three reasons:

- A `throw` statement (§14.18) was executed.
- An abnormal execution condition was synchronously detected by the Java virtual machine.

Such conditions arise because:

- ♦ evaluation of an expression violates the normal semantics of the language (§15.6), such as an integer divide by zero.
- ♦ an error occurs while loading, linking, or initializing part of the program (§12.2, §12.3, §12.4); in this case, an instance of a subclass of `LinkageError` is thrown.
- ♦ an internal error or resource limitation prevents the Java virtual machine from implementing the semantics of the Java programming language; in this case, an instance of a subclass of `VirtualMachineError` is thrown.

These exceptions are not thrown at an arbitrary point in the program, but rather at a point where they are specified as a possible result of an expression evaluation or statement execution.

- An asynchronous exception occurred (§11.1.3).

11.1.3 Asynchronous Exceptions

Most exceptions occur synchronously as a result of an action by the thread in which they occur, and at a point in the program that is specified to possibly result in such an exception. An *asynchronous exception* is, by contrast, an exception that can potentially occur at any point in the execution of a program.

Asynchronous exceptions occur only as a result of:

- An invocation of the (deprecated) `stop` method of class `Thread` or `ThreadGroup`.
- An internal error in the Java virtual machine; in this case, the asynchronous exception that is thrown is an instance of a subclass of `InternalError` or `UnknownError`.

The Java SE platform permits a small but bounded amount of execution to occur before an asynchronous exception is thrown.

11.2 Compile-Time Checking of Exceptions

A Java compiler checks, at compile time, that a program contains handlers for *checked exceptions*, by analyzing which checked exception types can result from execution of a method or constructor.

For each checked exception which is a possible result, the `throws` clause for the method (§8.4.6) or constructor (§8.8.5) must mention the class of that exception or one of the superclasses of the class of that exception. This compile-time checking for the presence of exception handlers is designed to reduce the number of exceptions which are not properly handled.

The checked exception classes (§11.1.1) named in the `throws` clause are part of the contract between the implementor and user of the method or constructor. The `throws` clause of an overriding method may not specify that this method will result in throwing any checked exception which the overridden method is not permitted, by its `throws` clause, to throw (§8.4.8.3).

When interfaces are involved, more than one method declaration may be overridden by a single overriding declaration. In this case, the overriding declaration must have a `throws` clause that is compatible with all the overridden declarations (§9.4.1).

The unchecked exception classes (§11.1.1) are exempted from compile-time checking.

We say that a statement or expression *can throw* a checked exception type *E* if, according to the rules given below, the execution of the statement or expression can result in an exception of type *E* being thrown.

11.2.1 Exception Analysis of Expressions

A class instance creation expression (§15.9) can throw an exception type *E* iff either:

- The expression is a qualified class instance creation expression and the qualifying expression can throw *E*; or
- Some expression of the argument list can throw *E*; or

- E is determined to be an exception type of the `throws` clause of the constructor that is invoked (§15.12.2.6); or
- The class instance creation expression includes a *ClassBody*, and some instance initializer block or instance variable initializer expression in the *ClassBody* can throw E .

A method invocation expression (§15.12) can throw an exception type E iff either:

- The method to be invoked is of the form *Primary.Identifier* and the *Primary* expression can throw E ; or
- Some expression of the argument list can throw E ; or
- E is determined to be an exception type of the `throws` clause of the method that is invoked (§15.12.2.6).

For every other kind of expression, the expression can throw an exception type E iff one of its immediate subexpressions can throw E .

11.2.2 Exception Analysis of Statements

A `throw` statement (§14.18) whose thrown expression has static type E can throw E , or any exception type thrown by the thrown expression.

A `try` statement (§14.20) can throw an exception type E iff either:

- The `try` block can throw E and E is not assignable to any `catch` parameter of the `try` statement and either no `finally` block is present or the `finally` block can complete normally; or
- Some `catch` block of the `try` statement can throw E and either no `finally` block is present or the `finally` block can complete normally; or
- A `finally` block is present and can throw E .

An explicit constructor invocation statement (§8.8.7.1) can throw an exception type E iff either:

- Some subexpression of the constructor invocation's parameter list can throw E ; or
- E is determined to be an exception type of the `throws` clause of the constructor that is invoked (§15.12.2.6).

Any other statement S can throw an exception type E iff an expression or statement immediately contained in S can throw E .

11.2.3 Exception Checking

It is a compile-time error if a method or constructor body can throw some exception type E when E is a checked exception type and E is not a subtype of some type declared in the `throws` clause of the method or constructor.

It is a compile-time error if a class variable initializer (§8.3.2) or static initializer (§8.7) of a named class or interface can throw a checked exception type.

It is a compile-time error if an instance variable initializer or instance initializer of a named class can throw a checked exception type unless that exception type or one of its supertypes is explicitly declared in the `throws` clause of each constructor of its class and the class has at least one explicitly declared constructor.

It is a compile-time error if a catch clause catches checked exception type E_1 when the try block corresponding to the catch clause can throw E_2 and E_2 is not a subtype of E_1 , unless E_1 is a supertype of `Exception`.

It is a compile-time error if a catch clause catches checked exception type E_1 and a preceding catch block of the immediately enclosing try statement catches E_1 or a supertype of E_1 .

11.3 Run-Time Handling of an Exception

When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically-enclosing catch clause, if any, of a try statement (§14.20) that can handle the exception.

A statement or expression is *dynamically enclosed* by a catch clause if it appears within the try block of the try statement of which the catch clause is a part, or if the caller of the statement or expression is dynamically enclosed by the catch clause.

The caller of a statement or expression depends on where it occurs:

- If within a method, then the caller is the method invocation expression (§15.12) that was executed to cause the method to be invoked.
- If within a constructor or an instance initializer or the initializer for an instance variable, then the caller is the class instance creation expression (§15.9) or the method invocation of `newInstance` that was executed to cause an object to be created.

- If within a static initializer or an initializer for a `static` variable, then the caller is the expression that used the class or interface so as to cause it to be initialized (§12.4).

Whether a particular catch clause *can handle* an exception is determined by comparing the class of the object that was thrown to the declared type of the parameter of the catch clause. The catch clause can handle the exception if the type of its parameter is the class of the exception or a superclass of the class of the exception.

The control transfer that occurs when an exception is thrown causes abrupt completion of expressions (§15.6) and statements (§14.1) until a catch clause is encountered that can handle the exception; execution then continues by executing the block of that catch clause. The code that caused the exception is never resumed.

All exceptions (synchronous and asynchronous) are *precise*: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated.

If no catch clause that can handle an exception can be found, then the current thread (the thread that encountered the exception) is terminated. Before termination, all `finally` clauses are executed and the uncaught exception is handled according to the following rules:

- If the current thread has an uncaught exception handler set, then that handler is executed.
- Otherwise, the method `uncaughtException` is invoked for the `ThreadGroup` that is the parent of the current thread. If the `ThreadGroup` and its parent `ThreadGroups` do not override `uncaughtException`, then the default handler's `uncaughtException` method is invoked.

Execution

THIS chapter specifies activities that occur during execution of a program. It is organized around the life cycle of the Java virtual machine and of the classes, interfaces, and objects that form a program.

The Java virtual machine starts up by loading a specified class and then invoking the method `main` in this specified class. Section §12.1 outlines the loading, linking, and initialization steps involved in executing `main`, as an introduction to the concepts in this chapter. Further sections specify the details of loading (§12.2), linking (§12.3), and initialization (§12.4).

The chapter continues with a specification of the procedures for creation of new class instances (§12.5); and finalization of class instances (§12.6). It concludes by describing the unloading of classes (§12.7) and the procedure followed when a program exits (§12.8).

12.1 Java virtual machine Start-Up

A Java virtual machine starts execution by invoking the method `main` of some specified class, passing it a single argument, which is an array of strings. In the examples in this specification, this first class is typically called `Test`.

The precise semantics of Java virtual machine start-up are given in chapter 5 of *The Java Virtual Machine Specification*. Here we present an overview of the process from the viewpoint of the Java programming language.

The manner in which the initial class is specified to the Java virtual machine is beyond the scope of this specification, but it is typical, in host environments that use command lines, for the fully-qualified name of the class to be specified as a command-line argument and for following command-line arguments to be used as strings to be provided as the argument to the method `main`.

We now outline the steps the Java virtual machine may take to execute `Test`, as an example of the loading, linking, and initialization processes that are described further in later sections.

12.1.1 Load the Class Test

The initial attempt to execute the method `main` of class `Test` discovers that the class `Test` is not loaded - that is, that the Java virtual machine does not currently contain a binary representation for this class. The Java virtual machine then uses a class loader to attempt to find such a binary representation. If this process fails, then an error is thrown. This loading process is described further in §12.2.

12.1.2 Link Test: Verify, Prepare, (Optionally) Resolve

After `Test` is loaded, it must be initialized before `main` can be invoked. And `Test`, like all (class or interface) types, must be linked before it is initialized. Linking involves verification, preparation, and (optionally) resolution. Linking is described further in §12.3.

Verification checks that the loaded representation of `Test` is well-formed, with a proper symbol table. Verification also checks that the code that implements `Test` obeys the semantic requirements of the Java programming language and the Java virtual machine. If a problem is detected during verification, then an error is thrown. Verification is described further in §12.3.1.

Preparation involves allocation of static storage and any data structures that are used internally by the implementation of the Java virtual machine, such as method tables. Preparation is described further in §12.3.2.

Resolution is the process of checking symbolic references from `Test` to other classes and interfaces, by loading the other classes and interfaces that are mentioned and checking that the references are correct.

The resolution step is optional at the time of initial linkage. An implementation may resolve symbolic references from a class or interface that is being linked very early, even to the point of resolving all symbolic references from the classes and interfaces that are further referenced, recursively. (This resolution may result in errors from these further loading and linking steps.) This implementation choice represents one extreme and is similar to the kind of "static" linkage that has been done for many years in simple implementations of the C language. (In these implementations, a compiled program is typically represented as an "a.out" file that contains a fully-linked version of the program, including completely resolved links to library

routines used by the program. Copies of these library routines are included in the "a.out" file.)

An implementation may instead choose to resolve a symbolic reference only when it is actively used; consistent use of this strategy for all symbolic references would represent the "laziest" form of resolution. In this case, if `Test` had several symbolic references to another class, then the references might be resolved one at a time, as they are used, or perhaps not at all, if these references were never used during execution of the program.

The only requirement on when resolution is performed is that any errors detected during resolution must be thrown at a point in the program where some action is taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error. Using the "static" example implementation choice described above, loading and linkage errors could occur before the program is executed if they involved a class or interface mentioned in the class `Test` or any of the further, recursively referenced, classes and interfaces. In a system that implemented the "laziest" resolution, these errors would be thrown only when an incorrect symbolic reference is actively used.

The resolution process is described further in §12.3.3.

12.1.3 Initialize Test: Execute Initializers

In our continuing example, the Java virtual machine is still trying to execute the method `main` of class `Test`. This is permitted only if the class has been initialized (§12.4.1).

Initialization consists of execution of any class variable initializers and static initializers of the class `Test`, in textual order. But before `Test` can be initialized, its direct superclass must be initialized, as well as the direct superclass of its direct superclass, and so on, recursively. In the simplest case, `Test` has `Object` as its implicit direct superclass; if class `Object` has not yet been initialized, then it must be initialized before `Test` is initialized. Class `Object` has no superclass, so the recursion terminates here.

If class `Test` has another class `Super` as its superclass, then `Super` must be initialized before `Test`. This requires loading, verifying, and preparing `Super` if this has not already been done and, depending on the implementation, may also involve resolving the symbolic references from `Super` and so on, recursively.

Initialization may thus cause loading, linking, and initialization errors, including such errors involving other types.

The initialization process is described further in §12.4.

12.1.4 Invoke `Test.main`

Finally, after completion of the initialization for class `Test` (during which other consequential loading, linking, and initializing may have occurred), the method `main` of `Test` is invoked.

The method `main` must be declared `public`, `static`, and `void`. It must accept a single argument that is an array of `String`. This method can be declared as either:

```
public static void main(String[] args)
```

or

```
public static void main(String... args)
```

12.2 Loading of Classes and Interfaces

Loading refers to the process of finding the binary form of a class or interface type with a particular name, perhaps by computing it on the fly, but more typically by retrieving a binary representation previously computed from source code by a Java compiler, and constructing, from that binary form, a `Class` object to represent the class or interface.

The precise semantics of loading are given in chapter 5 of *The Java Virtual Machine Specification*. Here we present an overview of the process from the viewpoint of the Java programming language.

The binary format of a class or interface is normally the `class` file format described in *The Java Virtual Machine Specification* cited above, but other formats are possible, provided they meet the requirements specified in §13.1. The method `defineClass` of class `ClassLoader` may be used to construct `Class` objects from binary representations in the `class` file format.

Well-behaved class loaders maintain these properties:

- Given the same name, a good class loader should always return the same class object.
- If a class loader *L1* delegates loading of a class *C* to another loader *L2*, then for any type *T* that occurs as the direct superclass or a direct superinterface of *C*, or as the type of a field in *C*, or as the type of a formal parameter of a method or

constructor in *C*, or as a return type of a method in *C*, *L1* and *L2* should return the same *Class* object.

A malicious class loader could violate these properties. However, it could not undermine the security of the type system, because the Java virtual machine guards against this.

12.2.1 The Loading Process

The loading process is implemented by the class *ClassLoader* and its subclasses. Different subclasses of *ClassLoader* may implement different loading policies. In particular, a class loader may cache binary representations of classes and interfaces, prefetch them based on expected usage, or load a group of related classes together. These activities may not be completely transparent to a running application if, for example, a newly compiled version of a class is not found because an older version is cached by a class loader. It is the responsibility of a class loader, however, to reflect loading errors only at points in the program where they could have arisen without prefetching or group loading.

If an error occurs during class loading, then an instance of one of the following subclasses of class *LinkageError* will be thrown at any point in the program that (directly or indirectly) uses the type:

- *ClassCircularityError*: A class or interface could not be loaded because it would be its own superclass or superinterface (§13.4.4).
- *ClassFormatError*: The binary data that purports to specify a requested compiled class or interface is malformed.
- *NoClassDefFoundError*: No definition for a requested class or interface could be found by the relevant class loader.

Because loading involves the allocation of new data structures, it may fail with an *OutOfMemoryError*.

12.3 Linking of Classes and Interfaces

Linking is the process of taking a binary form of a class or interface type and combining it into the runtime state of the Java virtual machine, so that it can be executed. A class or interface type is always loaded before it is linked.

Three different activities are involved in linking: verification, preparation, and resolution of symbolic references.

The precise semantics of linking are given in chapter 5 of *The Java Virtual Machine Specification*. Here we present an overview of the process from the viewpoint of the Java programming language.

This specification allows an implementation flexibility as to when linking activities (and, because of recursion, loading) take place, provided that the semantics of the language are respected, that a class or interface is completely verified and prepared before it is initialized, and that errors detected during linkage are thrown at a point in the program where some action is taken by the program that might require linkage to the class or interface involved in the error.

For example, an implementation may choose to resolve each symbolic reference in a class or interface individually, only when it is used (lazy or late resolution), or to resolve them all at once while the class is being verified (static resolution). This means that the resolution process may continue, in some implementations, after a class or interface has been initialized.

Because linking involves the allocation of new data structures, it may fail with an `OutOfMemoryError`.

12.3.1 Verification of the Binary Representation

Verification ensures that the binary representation of a class or interface is structurally correct. For example, it checks that every instruction has a valid operation code; that every branch instruction branches to the start of some other instruction, rather than into the middle of an instruction; that every method is provided with a structurally correct signature; and that every instruction obeys the type discipline of the Java virtual machine language.

If an error occurs during verification, then an instance of the following subclass of class `LinkageError` will be thrown at the point in the program that caused the class to be verified:

- `VerifyError`: The binary definition for a class or interface failed to pass a set of required checks to verify that it obeys the semantics of the Java virtual machine language and that it cannot violate the integrity of the Java virtual machine. (See §13.4.2, §13.4.4, §13.4.9, and §13.4.17 for some examples.)

12.3.2 Preparation of a Class or Interface Type

Preparation involves creating the `static` fields (class variables and constants) for a class or interface and initializing such fields to the default values (§4.12.5). This

does not require the execution of any source code; explicit initializers for static fields are executed as part of initialization (§12.4), not preparation.

12.3.3 Resolution of Symbolic References

The binary representation of a class or interface references other classes and interfaces and their fields, methods, and constructors symbolically, using the binary names (§13.1) of the other classes and interfaces (§13.1). For fields and methods, these symbolic references include the name of the class or interface type of which the field or method is a member, as well as the name of the field or method itself, together with appropriate type information.

Before a symbolic reference can be used it must undergo resolution, wherein a symbolic reference is checked to be correct and, typically, replaced with a direct reference that can be more efficiently processed if the reference is used repeatedly.

If an error occurs during resolution, then an error will be thrown. Most typically, this will be an instance of one of the following subclasses of the class `IncompatibleClassChangeError`, but it may also be an instance of some other subclass of `IncompatibleClassChangeError` or even an instance of the class `IncompatibleClassChangeError` itself. This error may be thrown at any point in the program that uses a symbolic reference to the type, directly or indirectly:

- `IllegalAccessError`: A symbolic reference has been encountered that specifies a use or assignment of a field, or invocation of a method, or creation of an instance of a class, to which the code containing the reference does not have access because the field or method was declared with `private`, `protected`, or default access (not `public`), or because the class was not declared `public`.

This can occur, for example, if a field that is originally declared `public` is changed to be `private` after another class that refers to the field has been compiled (§13.4.7).

- `InstantiationError`: A symbolic reference has been encountered that is used in class instance creation expression, but an instance cannot be created because the reference turns out to refer to an interface or to an abstract class.

This can occur, for example, if a class that is originally not `abstract` is changed to be `abstract` after another class that refers to the class in question has been compiled (§13.4.1).

- `NoSuchFieldError`: A symbolic reference has been encountered that refers to a specific field of a specific class or interface, but the class or interface does not contain a field of that name.

This can occur, for example, if a field declaration was deleted from a class after another class that refers to the field was compiled (§13.4.8).

- `NoSuchMethodError`: A symbolic reference has been encountered that refers to a specific method of a specific class or interface, but the class or interface does not contain a method of that signature.

This can occur, for example, if a method declaration was deleted from a class after another class that refers to the method was compiled (§13.4.12).

Additionally, an `UnsatisfiedLinkError` (a subclass of `LinkageError`) may be thrown if a class declares a native method for which no implementation can be found. The error will occur if the method is used, or earlier, depending on what kind of resolution strategy is being used by an implementation of the Java virtual machine (§12.3).

12.4 Initialization of Classes and Interfaces

Initialization of a class consists of executing its static initializers and the initializers for `static` fields (class variables) declared in the class.

Initialization of an interface consists of executing the initializers for fields (constants) declared in the interface.

Before a class is initialized, its direct superclass must be initialized, but interfaces implemented by the class are not initialized. Similarly, the superinterfaces of an interface are not initialized before the interface is initialized.

12.4.1 When Initialization Occurs

A class or interface type *T* will be initialized immediately before the first occurrence of any one of the following:

- *T* is a class and an instance of *T* is created.
- *T* is a class and a `static` method declared by *T* is invoked.
- A `static` field declared by *T* is assigned.
- A `static` field declared by *T* is used and the field is not a constant variable (§4.12.4).
- *T* is a top-level class, and an `assert` statement (§14.10) lexically nested within *T* is executed.

Invocation of certain reflective methods in class `Class` and in package `java.lang.reflect` also causes class or interface initialization. A class or interface will not be initialized under any other circumstance.

The intent here is that a class or interface type has a set of initializers that put it in a consistent state, and that this state is the first state that is observed by other classes. The static initializers and class variable initializers are executed in textual order, and may not refer to class variables declared in the class whose declarations appear textually after the use, even though these class variables are in scope (§8.3.2.3). This restriction is designed to detect, at compile time, most circular or otherwise malformed initializations.

The fact that initialization code is unrestricted allows examples to be constructed (§8.3.2.3) where the value of a class variable can be observed when it still has its initial default value, before its initializing expression is evaluated, but such examples are rare in practice. (Such examples can be also constructed for instance variable initialization; see the example at the end of §12.5). The full power of the language is available in these initializers; programmers must exercise some care. This power places an extra burden on code generators, but this burden would arise in any case because the language is concurrent (§12.4.2).

Before a class is initialized, its superclasses are initialized, if they have not previously been initialized.

A reference to a class field causes initialization of only the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a subinterface, or a class that implements an interface.

Initialization of an interface does not, of itself, cause initialization of any of its superinterfaces.

12.4.2 Detailed Initialization Procedure

Because the Java programming language is multithreaded, initialization of a class or interface requires careful synchronization, since some other thread may be trying to initialize the same class or interface at the same time. There is also the possibility that initialization of a class or interface may be requested recursively as part of the initialization of that class or interface; for example, a variable initializer in class *A* might invoke a method of an unrelated class *B*, which might in turn invoke a method of class *A*. The implementation of the Java virtual machine is responsible for taking care of synchronization and recursive initialization by using the following procedure. It assumes that the `Class` object has already been verified and prepared, and that the `Class` object contains state that indicates one of four situations:

- This `Class` object is verified and prepared but not initialized.
- This `Class` object is being initialized by some particular thread `T`.
- This `Class` object is fully initialized and ready for use.
- This `Class` object is in an erroneous state, perhaps because initialization was attempted and failed.

For each class or interface `C`, there is a unique initialization lock `LC`. The mapping from `C` to `LC` is left to the discretion of the Java virtual machine implementation. The procedure for initializing `C` is then as follows:

1. Synchronize on the initialization lock, `LC`, for `C`. This involves waiting until the current thread can acquire `LC`.
2. If the `Class` object for `C` indicates that initialization is in progress for `C` by some other thread, then release `LC` and block the current thread until informed that the in-progress initialization has completed, at which time repeat this step.
3. If the `Class` object for `C` indicates that initialization is in progress for `C` by the current thread, then this must be a recursive request for initialization. Release `LC` and complete normally.
4. If the `Class` object for `C` indicates that `C` has already been initialized, then no further action is required. Release `LC` and complete normally.
5. If the `Class` object for `C` is in an erroneous state, then initialization is not possible. Release `LC` and throw a `NoClassDefFoundError`.
6. Otherwise, record the fact that initialization of the `Class` object for `C` is in progress by the current thread, and release `LC`.

Then, initialize the final class variables and fields of interfaces whose values are compile-time constants (§8.3.2.1, §9.3.1, §13.4.9).

7. Next, if `C` is a class rather than an interface, and its superclass `SC` has not yet been initialized, then recursively perform this entire procedure for `SC`. If necessary, verify and prepare `SC` first. If the initialization of `SC` completes abruptly because of a thrown exception, then acquire `LC`, label the `Class` object for `C` as erroneous, notify all waiting threads, release `LC`, and complete abruptly, throwing the same exception that resulted from initializing `SC`.
8. Next, determine whether assertions are enabled (§14.10) for `C` by querying its defining class loader.

9. Next, execute either the class variable initializers and static initializers of the class, or the field initializers of the interface, in textual order, as though they were a single block.
10. If the execution of the initializers completes normally, then acquire *LC*, label the `Class` object for *C* as fully initialized, notify all waiting threads, release *LC*, and complete this procedure normally.
11. Otherwise, the initializers must have completed abruptly by throwing some exception *E*. If the class of *E* is not `Error` or one of its subclasses, then create a new instance of the class `ExceptionInInitializerError`, with *E* as the argument, and use this object in place of *E* in the following step. But if a new instance of `ExceptionInInitializerError` cannot be created because an `OutOfMemoryError` occurs, then instead use an `OutOfMemoryError` object in place of *E* in the following step.
12. Acquire *LC*, label the `Class` object for *C* as erroneous, notify all waiting threads, release *LC*, and complete this procedure abruptly with reason *E* or its replacement as determined in the previous step.

12.5 Creation of New Class Instances

A new class instance is explicitly created when evaluation of a class instance creation expression (§15.9) causes a class to be instantiated.

A new class instance may be implicitly created in the following situations:

- Loading of a class or interface that contains a `String` literal (§3.10.5) may create a new `String` object to represent that literal. (This might not occur if the same `String` has previously been interned (§3.10.5).)
- Execution of an operation that causes boxing conversion (§5.1.7). Boxing conversion may create a new object of a wrapper class associated with one of the primitive types.
- Execution of a string concatenation operator (§15.18.1) that is not part of a constant expression sometimes creates a new `String` object to represent the result. String concatenation operators may also create temporary wrapper objects for a value of a primitive type.

Each of these situations identifies a particular constructor (§8.8) to be called with specified arguments (possibly none) as part of the class instance creation process.

Whenever a new class instance is created, memory space is allocated for it with room for all the instance variables declared in the class type and all the instance variables declared in each superclass of the class type, including all the instance variables that may be hidden (§8.3).

If there is not sufficient space available to allocate memory for the object, then creation of the class instance completes abruptly with an `OutOfMemoryError`. Otherwise, all the instance variables in the new object, including those declared in superclasses, are initialized to their default values (§4.12.5).

Just before a reference to the newly created object is returned as the result, the indicated constructor is processed to initialize the new object using the following procedure:

1. Assign the arguments for the constructor to newly created parameter variables for this constructor invocation.
2. If this constructor begins with an explicit constructor invocation (§8.8.7.1) of another constructor in the same class (using `this`), then evaluate the arguments and process that constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason; otherwise, continue with step 5.
3. This constructor does not begin with an explicit constructor invocation of another constructor in the same class (using `this`). If this constructor is for a class other than `Object`, then this constructor will begin with an explicit or implicit invocation of a superclass constructor (using `super`). Evaluate the arguments and process that superclass constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, continue with step 4.
4. Execute the instance initializers and instance variable initializers for this class, assigning the values of instance variable initializers to the corresponding instance variables, in the left-to-right order in which they appear textually in the source code for the class. If execution of any of these initializers results in an exception, then no further initializers are processed and this procedure completes abruptly with that same exception. Otherwise, continue with step 5.
5. Execute the rest of the body of this constructor. If that execution completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, this procedure completes normally.

Unlike C++, the Java programming language does not specify altered rules for method dispatch during the creation of a new class instance. If methods are

invoked that are overridden in subclasses in the object being initialized, then these overriding methods are used, even before the new object is completely initialized.

12.6 Finalization of Class Instances

The class `Object` has a protected method called `finalize`; this method can be overridden by other classes. The particular definition of `finalize` that can be invoked for an object is called the *finalizer* of that object. Before the storage for an object is reclaimed by the garbage collector, the Java virtual machine will invoke the finalizer of that object.

Finalizers provide a chance to free up resources that cannot be freed automatically by an automatic storage manager. In such situations, simply reclaiming the memory used by an object would not guarantee that the resources it held would be reclaimed.

The Java programming language does not specify how soon a finalizer will be invoked, except to say that it will happen before the storage for the object is reused. Also, the language does not specify which thread will invoke the finalizer for any given object.

It is guaranteed that the thread that invokes the finalizer will not be holding any user-visible synchronization locks when the finalizer is invoked.

If an uncaught exception is thrown during the finalization, the exception is ignored and finalization of that object terminates.

The completion of an object's constructor happens-before (§17.4.5) the execution of its `finalize` method (in the formal sense of happens-before).

The `finalize` method declared in class `Object` takes no action. The fact that class `Object` declares a `finalize` method means that the `finalize` method for any class can always invoke the `finalize` method for its superclass. This should always be done, unless it is the programmer's intent to nullify the actions of the finalizer in the superclass. (Unlike constructors, finalizers do not automatically invoke the finalizer for the superclass; such an invocation must be coded explicitly.)

A finalizer may be invoked explicitly, just like any other method.

The package `java.lang.ref` describes weak references, which interact with garbage collection and finalization. As with any API that has special interactions with the language, implementors must be cognizant of any requirements imposed by the `java.lang.ref` API. This specification does not discuss weak references in any way. Readers are referred to the API documentation for details.

12.6.1 Implementing Finalization

Every object can be characterized by two attributes: it may be *reachable*, *finalizer-reachable*, or *unreachable*, and it may also be *unfinalized*, *finalizable*, or *finalized*.

A *reachable* object is any object that can be accessed in any potential continuing computation from any live thread.

A *finalizer-reachable* object can be reached from some finalizable object through some chain of references, but not from any live thread.

An *unreachable* object cannot be reached by either means.

An *unfinalized* object has never had its finalizer automatically invoked.

A *finalized* object has had its finalizer automatically invoked.

A *finalizable* object has never had its finalizer automatically invoked, but the Java virtual machine may eventually automatically invoke its finalizer.

An object *o* is not finalizable until its constructor has invoked the constructor for `Object` on *o* and that invocation has completed successfully (that is, without throwing an exception). Every pre-finalization write to a field of an object must be visible to the finalization of that object. Furthermore, none of the pre-finalization reads of fields of that object may see writes that occur after finalization of that object is initiated.

12.6.1.1 Interaction with the Memory Model

It must be possible for the memory model (§17.4) to decide when it can commit actions that take place in a finalizer. This section describes the interaction of finalization with the memory model.

Each execution has a number of *reachability decision points*, labeled *di*. Each action either *comes-before di* or *comes-after di*. Other than as explicitly mentioned, the comes-before ordering described in this section is unrelated to all other orderings in the memory model.

If *r* is a read that sees a write *w* and *r* comes-before *di*, then *w* must come-before *di*.

If *x* and *y* are synchronization actions on the same variable or monitor such that *so*(*x*, *y*) (§17.4.4) and *y* comes-before *di*, then *x* must come-before *di*.

At each reachability decision point, some set of objects are marked as unreachable, and some subset of those objects are marked as finalizable. These reachability decision points are also the points at which references are checked, enqueued, and

cleared according to the rules provided in the API documentation for the package `java.lang.ref`.

The only objects that are considered definitely reachable at a point di are those that can be shown to be reachable by the application of these rules:

- An object B is definitely reachable at di from `static` fields if there exists a write $w1$ to a `static` field v of a class C such that the value written by $w1$ is a reference to B , the class C is loaded by a reachable classloader, and there does not exist a write $w2$ to v such that $hb(w2, w1)$ is not true and both $w1$ and $w2$ come-before di .
- An object B is definitely reachable from A at di if there is a write $w1$ to an element v of A such that the value written by $w1$ is a reference to B and there does not exist a write $w2$ to v such that $hb(w2, w1)$ is not true and both $w1$ and $w2$ come-before di .
- If an object C is definitely reachable from an object B , and object B is definitely reachable from an object A , then C is definitely reachable from A .

An action a is an active use of X if and only if at least one of the following conditions holds:

- a reads or writes an element of X
- a locks or unlocks X and there is a lock action on X that happens-after the invocation of the finalizer for X
- a writes a reference to X
- a is an active use of an object Y , and X is definitely reachable from Y

If an object X is marked as unreachable at di , then:

- X must not be definitely reachable at di from `static` fields; and
- All active uses of X in thread t that come-after di must occur in the finalizer invocation for X or as a result of thread t performing a read that comes-after di of a reference to X ; and
- All reads that come-after di that see a reference to X must see writes to elements of objects that were unreachable at di , or see writes that came-after di .

If an object X is marked as finalizable at di , then:

- X must be marked as unreachable at di ; and
- di must be the only place where X is marked as finalizable; and
- actions that happen-after the finalizer invocation must come-after di .

12.6.2 Finalizer Invocations are Not Ordered

The Java programming language imposes no ordering on `finalize` method calls. Finalizers may be called in any order, or even concurrently.

12.7 Unloading of Classes and Interfaces

An implementation of the Java programming language may *unload* classes.

A class or interface may be unloaded if and only if its defining class loader may be reclaimed by the garbage collector as discussed in §12.6.

Classes and interfaces loaded by the bootstrap loader may not be unloaded.

12.8 Program Exit

A program terminates all its activity and *exits* when one of two things happens:

- All the threads that are not daemon threads terminate.
- Some thread invokes the `exit` method of class `Runtime` or class `System`, and the `exit` operation is not forbidden by the security manager.

Binary Compatibility

DEVELOPMENT tools for the Java programming language should support automatic recompilation as necessary whenever source code is available. Particular implementations may also store the source and binary of types in a versioning database and implement a `ClassLoader` that uses integrity mechanisms of the database to prevent linkage errors by providing binary-compatible versions of types to clients.

Developers of packages and classes that are to be widely distributed face a different set of problems. In the Internet, which is our favorite example of a widely distributed system, it is often impractical or impossible to automatically recompile the pre-existing binaries that directly or indirectly depend on a type that is to be changed. Instead, this specification defines a set of changes that developers are permitted to make to a package or to a class or interface type while preserving (not breaking) compatibility with pre-existing binaries.

The paper quoted above appears in *Proceedings of OOPSLA '95*, published as *ACM SIGPLAN Notices*, Volume 30, Number 10, October 1995, pages 426-438. Within the framework of that paper, Java programming language binaries are binary compatible under all relevant transformations that the authors identify (with some caveats with respect to the addition of instance variables). Using their scheme, here is a list of some important binary compatible changes that the Java programming language supports:

- Reimplementing existing methods, constructors, and initializers to improve performance.
- Changing methods or constructors to return values on inputs for which they previously either threw exceptions that normally should not occur or failed by going into an infinite loop or causing a deadlock.
- Adding new fields, methods, or constructors to an existing class or interface.
- Deleting private fields, methods, or constructors of a class.

- When an entire package is updated, deleting default (package-only) access fields, methods, or constructors of classes and interfaces in the package.
- Reordering the fields, methods, or constructors in an existing type declaration.
- Moving a method upward in the class hierarchy.
- Reordering the list of direct superinterfaces of a class or interface.
- Inserting new class or interface types in the type hierarchy.

This chapter specifies minimum standards for binary compatibility guaranteed by all implementations. The Java programming language guarantees compatibility when binaries of classes and interfaces are mixed that are not known to be from compatible sources, but whose sources have been modified in the compatible ways described here. Note that we are discussing compatibility between releases of an application. A discussion of compatibility among releases of the Java SE platform is beyond the scope of this chapter.

We encourage development systems to provide facilities that alert developers to the impact of changes on pre-existing binaries that cannot be recompiled.

This chapter first specifies some properties that any binary format for the Java programming language must have (§13.1). It next defines binary compatibility, explaining what it is and what it is not (§13.2). It finally enumerates a large set of possible changes to packages (§13.3), classes (§13.4), and interfaces (§13.5), specifying which of these changes are guaranteed to preserve binary compatibility and which are not.

13.1 The Form of a Binary

Programs must be compiled either into the `class` file format specified by the *The Java Virtual Machine Specification*, or into a representation that can be mapped into that format by a class loader written in the Java programming language. Furthermore, the resulting `class` file must have certain properties. A number of these properties are specifically chosen to support source code transformations that preserve binary compatibility.

The required properties are:

1. The class or interface must be named by its *binary name*, which must meet the following constraints:
 - The binary name of a top-level type is its canonical name (§6.7).

- The binary name of a member type consists of the binary name of its immediately enclosing type, followed by \$, followed by the simple name of the member.
 - The binary name of a local class (§14.3) consists of the binary name of its immediately enclosing type, followed by \$, followed by a non-empty sequence of digits, followed by the simple name of the local class.
 - The binary name of an anonymous class (§15.9.5) consists of the binary name of its immediately enclosing type, followed by \$, followed by a non-empty sequence of digits.
 - The binary name of a type variable declared by a generic class or interface is the binary name of its immediately enclosing type, followed by \$, followed by the simple name of the type variable.
 - The binary name of a type variable declared by a generic method is the binary name of the type declaring the method, followed by \$, followed by the descriptor of the method as defined in *The Java Virtual Machine Specification*, followed by \$, followed by the simple name of the type variable.
 - The binary name of a type variable declared by a generic constructor is the binary name of the type declaring the constructor, followed by \$, followed by the descriptor of the constructor as defined in *The Java Virtual Machine Specification*, followed by \$, followed by the simple name of the type variable.
2. A reference to another class or interface type must be symbolic, using the binary name of the type.
 3. References to fields that are constant variables (§4.12.4) are resolved at compile time to the constant value that is denoted. No reference to such a field should be present in the code in a binary file (except in the class or interface containing the field, which will have code to initialize it). Such a field must always appear to have been initialized (§12.4.2); the default initial value for the type of such a field must never be observed. See §13.4.9 for a discussion.
 4. Given a legal expression denoting a field access in a class *C*, referencing a non-constant (§13.4.9) field named *f* declared in a (possibly distinct) class or interface *D*, we define the *qualifying type of the field reference* as follows:
 - If the expression is of the form *Primary*. *f* then:
 - ♦ If the compile-time type of *Primary* is an intersection type (§4.9) $V_1 \& \dots \& V_n$, then the qualifying type of the reference is V_1 .

- ♦ Otherwise, the compile-time type of *Primary* is the qualifying type of the reference.
- If the expression is of the form *super.f* then the superclass of *C* is the qualifying type of the reference.
- If the expression is of the form *X.super.f* then the superclass of *X* is the qualifying type of the reference.
- If the reference is of the form *X.f*, where *X* denotes a class or interface, then the class or interface denoted by *X* is the qualifying type of the reference.
- If the expression is referenced by a simple name, then if *f* is a member of the current class or interface, *C*, then let *T* be *C*. Otherwise, let *T* be the innermost lexically enclosing class of which *f* is a member. In either case, *T* is the qualifying type of the reference.

The reference to *f* must be compiled into a symbolic reference to the erasure (§4.6) of the qualifying type of the reference, plus the simple name of the field, *f*. The reference must also include a symbolic reference to the erasure of the declared type of the field so that the verifier can check that the type is as expected.

5. Given a method invocation expression in a class or interface *C* referencing a method named *m* declared (or implicitly declared (§9.2)) in a (possibly distinct) class or interface *D*, we define the *qualifying type of the method invocation* as follows:

If *D* is `Object` then the qualifying type of the expression is `Object`. Otherwise:

- If the expression is of the form *Primary.m* then:
 - ♦ If the compile-time type of *Primary* is an intersection type (§4.9) $V_1 \& \dots \& V_n$, then the qualifying type of the method invocation is V_1 .
 - ♦ Otherwise, the compile-time type of *Primary* is the qualifying type of the method invocation.
- If the expression is of the form *super.m* then the superclass of *C* is the qualifying type of the method invocation.
- If the expression is of the form *X.super.m* then the superclass of *X* is the qualifying type of the method invocation.
- If the reference is of the form *X.m*, where *X* denotes a class or interface, then the class or interface denoted by *X* is the qualifying type of the method invocation.

- If the method is referenced by a simple name, then if m is a member of the current class or interface, C , then let T be C . Otherwise, let T be the innermost lexically enclosing class of which m is a member. In either case, T is the qualifying type of the method invocation.

A reference to a method must be resolved at compile time to a symbolic reference to the erasure (§4.6) of the qualifying type of the invocation, plus the erasure of the signature of the method (§8.4.2). A reference to a method must also include either a symbolic reference to the erasure of the return type of the denoted method or an indication that the denoted method is declared void and does not return a value. The signature of a method must include all of the following:

- The simple name of the method
 - The number of formal parameters of the method
 - A symbolic reference to the type of each formal parameter
6. Given a class instance creation expression (§15.9) or a constructor invocation statement (§8.8.7.1) in a class or interface C referencing a constructor m declared in a (possibly distinct) class or interface D , we define the qualifying type of the constructor invocation as follows:
- If the expression is of the form `new D(...)` or `X.new D(...)`, then the qualifying type of the invocation is D .
 - If the expression is of the form `new D(...){...}` or `X.new D(...){...}`, then the qualifying type of the expression is the compile-time type of the expression.
 - If the expression is of the form `super(...)` or `Primary.super(...)` then the qualifying type of the expression is the direct superclass of C .
 - If the expression is of the form `this(...)`, then the qualifying type of the expression is C .

A reference to a constructor must be resolved at compile time to a symbolic reference to the erasure (§4.6) of the qualifying type of the invocation, plus the signature of the constructor (§8.8.2). The signature of a constructor must include both:

- The number of parameters of the constructor
- A symbolic reference to the type of each formal parameter

In addition, the constructor of a non-private inner member class must be compiled such that it has as its first parameter, an additional implicit parameter representing the immediately enclosing instance (§8.1.3).

7. Any constructs introduced by a Java compiler that do not have a corresponding construct in the source code must be marked as synthetic, except for default constructors, the class initialization method, and the `values` and `valueOf` methods of the Enum class.

A binary representation for a class or interface must also contain all of the following:

1. If it is a class and is not class `Object`, then a symbolic reference to the erasure of the direct superclass of this class.
2. A symbolic reference to the erasure of each direct superinterface, if any.
3. A specification of each field declared in the class or interface, given as the simple name of the field and a symbolic reference to the erasure of the type of the field.
4. If it is a class, then the erased signature of each constructor, as described above.
5. For each method declared in the class or interface (excluding, for an interface, its implicitly declared methods (§9.2)), its erased signature and return type, as described above.
6. The code needed to implement the class or interface:
 - For an interface, code for the field initializers
 - For a class, code for the field initializers, the instance and static initializers, and the implementation of each method or constructor
7. Every type must contain sufficient information to recover its canonical name (§6.7).
8. Every member type must have sufficient information to recover its source level access modifier.
9. Every nested class must have a symbolic reference to its immediately enclosing class.
10. Every class that contains a nested class must contain symbolic references to all of its member classes, and to all local and anonymous classes that appear in its methods, constructors, and static or instance initializers.

The following sections discuss changes that may be made to class and interface type declarations without breaking compatibility with pre-existing binaries. Under the translation requirements given above, the Java virtual machine and its `class` file format support these changes. Any other valid binary format, such as a compressed or encrypted representation that is mapped back into `class` files by a class loader under the above requirements, will necessarily support these changes as well.

13.2 What Binary Compatibility Is and Is Not

A change to a type is *binary compatible with* (equivalently, does not *break binary compatibility with*) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error.

Binaries are compiled to rely on the accessible members and constructors of other classes and interfaces. To preserve binary compatibility, a class or interface should treat its accessible members and constructors, their existence and behavior, as a *contract* with its users.

The Java programming language is designed to prevent additions to contracts and accidental name collisions from breaking binary compatibility. Specifically, addition of more methods overloading a particular method name does not break compatibility with pre-existing binaries. The method signature that the pre-existing binary will use for method lookup is chosen by the method overload resolution algorithm at compile time (§15.12.2).

Binary compatibility is not the same as source compatibility. In particular, the example in §13.4.6 shows that a set of compatible binaries can be produced from sources that will not compile all together. This example is typical: a new declaration is added, changing the meaning of a name in an unchanged part of the source code, while the pre-existing binary for that unchanged part of the source code retains the fully-qualified, previous meaning of the name. Producing a consistent set of source code requires providing a qualified name or field access expression corresponding to the previous meaning.

13.3 Evolution of Packages

A new top-level class or interface type may be added to a package without breaking compatibility with pre-existing binaries, provided the new type does not reuse a name previously given to an unrelated type.

If a new type reuses a name previously given to an unrelated type, then a conflict may result, since binaries for both types could not be loaded by the same class loader.

Changes in top-level class and interface types that are not `public` and that are not a superclass or superinterface, respectively, of a `public` type, affect only types within the package in which they are declared. Such types may be deleted or otherwise changed, even if incompatibilities are otherwise described here, provided that the affected binaries of that package are updated together.

13.4 Evolution of Classes

This section describes the effects of changes to the declaration of a class and its members and constructors on pre-existing binaries.

13.4.1 `abstract` Classes

If a class that was not declared `abstract` is changed to be declared `abstract`, then pre-existing binaries that attempt to create new instances of that class will throw either an `InstantiationError` at link time, or (if a reflective method is used) an `InstantiationException` at run time; such a change is therefore not recommended for widely distributed classes.

Changing a class that is declared `abstract` to no longer be declared `abstract` does not break compatibility with pre-existing binaries.

13.4.2 `final` Classes

If a class that was not declared `final` is changed to be declared `final`, then a `VerifyError` is thrown if a binary of a pre-existing subclass of this class is loaded, because `final` classes can have no subclasses; such a change is not recommended for widely distributed classes.

Changing a class that is declared `final` to no longer be declared `final` does not break compatibility with pre-existing binaries.

13.4.3 `public` Classes

Changing a class that is not declared `public` to be declared `public` does not break compatibility with pre-existing binaries.

If a class that was declared `public` is changed to not be declared `public`, then an `IllegalAccessError` is thrown if a pre-existing binary is linked that needs but no longer has access to the class type; such a change is not recommended for widely distributed classes.

13.4.4 Superclasses and Superinterfaces

A `ClassCircularityError` is thrown at load time if a class would be a superclass of itself. Changes to the class hierarchy that could result in such a circularity when newly compiled binaries are loaded with pre-existing binaries are not recommended for widely distributed classes.

Changing the direct superclass or the set of direct superinterfaces of a class type will not break compatibility with pre-existing binaries, provided that the total set of superclasses or superinterfaces, respectively, of the class type loses no members.

If a change to the direct superclass or the set of direct superinterfaces results in any class or interface no longer being a superclass or superinterface, respectively, then linkage errors may result if pre-existing binaries are loaded with the binary of the modified class. Such changes are not recommended for widely distributed classes.

13.4.5 Class Type Parameters

Adding or removing a type parameter of a class does not, in itself, have any implications for binary compatibility.

If such a type parameter is used in the type of a field or method, that may have the normal implications of changing the aforementioned type.

Renaming a type parameter of a class has no effect with respect to pre-existing binaries.

Changing the first bound of a type parameter of a class may change the erasure (§4.6) of any member that uses that type parameter in its own type, and this may affect binary compatibility. The change of such a bound is analogous to the change of the first bound of a type parameter of a method or constructor (§13.4.13).

Changing any other bound has no effect on binary compatibility.

13.4.6 Class Body and Member Declarations

No incompatibility with pre-existing binaries is caused by adding an instance (respectively `static`) member that has the same name and accessibility (for fields), or same name and accessibility and signature and return type (for methods), as an

instance (respectively `static`) member of a superclass or subclass. No error occurs even if the set of classes being linked would encounter a compile-time error.

Deleting a class member or constructor that is not declared `private` may cause a linkage error if the member or constructor is used by a pre-existing binary.

The `super` keyword can be used to access a method declared in a superclass, bypassing any methods declared in the current class. The expression `super.Identifier` is resolved, at compile time, to a method M in the superclass S . If the method M is an instance method, then the method MR invoked at run-time is the method with the same signature as M that is a member of the direct superclass of the class containing the expression involving `super`.

13.4.7 Access to Members and Constructors

Changing the declared access of a member or constructor to permit less access may break compatibility with pre-existing binaries, causing a linkage error to be thrown when these binaries are resolved. Less access is permitted if the access modifier is changed from default access to `private` access; from `protected` access to default or `private` access; or from `public` access to `protected`, default, or `private` access. Changing a member or constructor to permit less access is therefore not recommended for widely distributed classes.

Perhaps surprisingly, the binary format is defined so that changing a member or constructor to be more accessible does not cause a linkage error when a subclass (already) defines a method to have less access.

Allowing superclasses to change `protected` methods to be `public` without breaking binaries of pre-existing subclasses helps make binaries less fragile. The alternative, where such a change would cause a linkage error, would create additional binary incompatibilities.

13.4.8 Field Declarations

Widely distributed programs should not expose any fields to their clients. Apart from the binary compatibility issues discussed below, this is generally good software engineering practice. Adding a field to a class may break compatibility with pre-existing binaries that are not recompiled.

Assume a reference to a field f with qualifying type T . Assume further that f is in fact an instance (respectively `static`) field declared in a superclass of T , S , and that the type of f is X .

If a new field of type X with the same name as f is added to a subclass of S that is a superclass of T or T itself, then a linkage error may occur. Such a linkage error will occur only if, in addition to the above, either one of the following conditions hold:

- The new field is less accessible than the old one.
- The new field is a `static` (respectively `instance`) field.

In particular, no linkage error will occur in the case where a class could no longer be recompiled because a field access previously referenced a field of a superclass with an incompatible type. The previously compiled class with such a reference will continue to reference the field declared in a superclass.

Deleting a field from a class will break compatibility with any pre-existing binaries that reference this field, and a `NoSuchFieldError` will be thrown when such a reference from a pre-existing binary is linked. Only `private` fields may be safely deleted from a widely distributed class.

For purposes of binary compatibility, adding or removing a field f whose type involves type variables (§4.4) or parameterized types (§4.5) is equivalent to the addition (respectively, removal) of a field of the same name whose type is the erasure (§4.6) of the type of f .

13.4.9 final Fields and Constants

If a field that was not declared `final` is changed to be declared `final`, then it can break compatibility with pre-existing binaries that attempt to assign new values to the field.

Deleting the keyword `final` or changing the value to which a field is initialized does not break compatibility with existing binaries.

If a field is a constant variable (§4.12.4), then deleting the keyword `final` or changing its value will not break compatibility with pre-existing binaries by causing them not to run, but they will not see any new value for the usage of the field unless they are recompiled. This is true even if the usage itself is not a compile-time constant expression (§15.28).

This result is a side-effect of the decision to support conditional compilation, as discussed at the end of §14.21.

The best way to avoid problems with "inconstant constants" in widely-distributed code is to declare as compile time constants only values which truly are unlikely ever to change. Other than for true mathematical constants, we recommend that source code make very sparing use of class variables that are declared `static` and

`final`. If the read-only nature of `final` is required, a better choice is to declare a `private static` variable and a suitable accessor method to get its value.

Thus we recommend:

```
private static int N;  
public static int getN() { return N; }
```

rather than:

```
public static final int N = ...;
```

There is no problem with:

```
public static int N = ...;
```

if `N` need not be read-only. We also recommend, as a general rule, that only truly constant values be declared in interfaces.

We note, but do not recommend, that if a field of primitive type of an interface may change, its value may be expressed idiomatically as in:

```
interface Flags {  
    boolean debug = new Boolean(true).booleanValue();  
}
```

ensuring that this value is not a constant. Similar idioms exist for the other primitive types.

One other thing to note is that `static final` fields that have constant values (whether of primitive or `String` type) must never appear to have the default initial value for their type (§4.12.5). This means that all such fields appear to be initialized first during class initialization (§8.3.2.1, §9.3.1, §12.4.2).

13.4.10 `static Fields`

If a field that is not declared `private` was not declared `static` and is changed to be declared `static`, or vice versa, then a linkage error, specifically an `IncompatibleClassChangeError`, will result if the field is used by a pre-existing binary which expected a field of the other kind. Such changes are not recommended in code that has been widely distributed.

13.4.11 `transient Fields`

Adding or deleting a `transient` modifier of a field does not break compatibility with pre-existing binaries.

13.4.12 Method and Constructor Declarations

Adding a method or constructor declaration to a class will not break compatibility with any pre-existing binaries, even in the case where a type could no longer be recompiled because an invocation previously referenced a method or constructor of a superclass with an incompatible type. The previously compiled class with such a reference will continue to reference the method or constructor declared in a superclass.

Assume a reference to a method *m* with qualifying type *T*. Assume further that *m* is in fact an instance (respectively `static`) method declared in a superclass of *T*, *S*.

If a new method of type *X* with the same signature and return type as *m* is added to a subclass of *S* that is a superclass of *T* or *T* itself, then a linkage error may occur. Such a linkage error will occur only if, in addition to the above, either one of the following conditions hold:

- The new method is less accessible than the old one.
- The new method is a `static` (respectively instance) method.

Deleting a method or constructor from a class may break compatibility with any pre-existing binary that referenced this method or constructor; a `NoSuchMethodError` may be thrown when such a reference from a pre-existing binary is linked. Such an error will occur only if no method with a matching signature and return type is declared in a superclass.

13.4.13 Method and Constructor Type Parameters

Adding or removing a type parameter of a method or constructor does not, in itself, have any implications for binary compatibility.

If such a type parameter is used in the type of the method or constructor, that may have the normal implications of changing the aforementioned type.

Renaming a type parameter of a method or constructor has no effect with respect to pre-existing binaries.

Changing the first bound of a type parameter of a method or constructor may change the erasure (§4.6) of any member that uses that type parameter in its own type, and this may affect binary compatibility. Specifically:

- If the type parameter is used as the type of a field, the effect is as if the field was removed and a field with the same name, whose type is the new erasure of the type variable, was added.

- If the type parameter is used as the type of any formal parameter of a method, but not as the return type, the effect is as if that method were removed, and replaced with a new method that is identical except for the types of the aforementioned formal parameters, which now have the new erasure of the type parameter as their type.
- If the type parameter is used as a return type of a method, but not as the type of any formal parameter of the method, the effect is as if that method were removed, and replaced with a new method that is identical except for the return type, which is now the new erasure of the type parameter.
- If the type parameter is used as a return type of a method and as the type of one or more formal parameters of the method, the effect is as if that method were removed, and replaced with a new method that is identical except for the return type, which is now the new erasure of the type parameter, and except for the types of the aforementioned formal parameters, which now have the new erasure of the type parameter as their types.

Changing any other bound has no effect on binary compatibility.

13.4.14 Method and Constructor Formal Parameters

Changing the name of a formal parameter of a method or constructor does not impact pre-existing binaries.

Changing the name of a method, or the type of a formal parameter to a method or constructor, or adding a parameter to or deleting a parameter from a method or constructor declaration creates a method or constructor with a new signature, and has the combined effect of deleting the method or constructor with the old signature and adding a method or constructor with the new signature (§13.4.12).

Changing the type of the last formal parameter of a method from $T[]$ to a variable arity parameter (§8.4.1) of type T (i.e. to $T \dots$), and vice versa, does not impact pre-existing binaries.

For purposes of binary compatibility, adding or removing a method or constructor m whose signature involves type variables (§4.4) or parameterized types (§4.5) is equivalent to the addition (respectively, removal) of an otherwise equivalent method whose signature is the erasure (§4.6) of the signature of m .

13.4.15 Method Result Type

Changing the result type of a method, or replacing a result type with `void`, or replacing `void` with a result type, has the combined effect of deleting the old

method and adding a new method with the new result type or newly void result (see §13.4.12).

For purposes of binary compatibility, adding or removing a method or constructor *m* whose return type involves type variables (§4.4) or parameterized types (§4.5) is equivalent to the addition (respectively, removal) of the an otherwise equivalent method whose return type is the erasure (§4.6) of the return type of *m*.

13.4.16 abstract Methods

Changing a method that is declared `abstract` to no longer be declared `abstract` does not break compatibility with pre-existing binaries.

Changing a method that is not declared `abstract` to be declared `abstract` will break compatibility with pre-existing binaries that previously invoked the method, causing an `AbstractMethodError`.

13.4.17 final Methods

Changing a method that is declared `final` to no longer be declared `final` does not break compatibility with pre-existing binaries.

Changing an instance method that is not declared `final` to be declared `final` may break compatibility with existing binaries that depend on the ability to override the method.

Changing a class (`static`) method that is not declared `final` to be declared `final` does not break compatibility with existing binaries, because the method could not have been overridden.

13.4.18 native Methods

Adding or deleting a `native` modifier of a method does not break compatibility with pre-existing binaries.

The impact of changes to types on pre-existing `native` methods that are not recompiled is beyond the scope of this specification and should be provided with the description of an implementation. Implementations are encouraged, but not required, to implement `native` methods in a way that limits such impact.

13.4.19 `static` Methods

If a method that is not declared `private` is also declared `static` (that is, a class method) and is changed to not be declared `static` (that is, to an instance method), or vice versa, then compatibility with pre-existing binaries may be broken, resulting in a linkage time error, namely an `IncompatibleClassChangeError`, if these methods are used by the pre-existing binaries. Such changes are not recommended in code that has been widely distributed.

13.4.20 `synchronized` Methods

Adding or deleting a `synchronized` modifier of a method does not break compatibility with pre-existing binaries.

13.4.21 Method and Constructor Throws

Changes to the `throws` clause of methods or constructors do not break compatibility with pre-existing binaries; these clauses are checked only at compile time.

13.4.22 Method and Constructor Body

Changes to the body of a method or constructor do not break compatibility with pre-existing binaries.

The keyword `final` on a method does not mean that the method can be safely inlined; it means only that the method cannot be overridden. It is still possible that a new version of that method will be provided at link time. Furthermore, the structure of the original program must be preserved for purposes of reflection.

Therefore, we note that a Java compiler cannot expand a method inline at compile time. In general we suggest that implementations use late-bound (run-time) code generation and optimization.

13.4.23 Method and Constructor Overloading

Adding new methods or constructors that overload existing methods or constructors does not break compatibility with pre-existing binaries. The signature to be used for each invocation was determined when these existing binaries were compiled; therefore newly added methods or constructors will not be used, even if their signatures are both applicable and more specific than the signature originally chosen.

While adding a new overloaded method or constructor may cause a compile-time error the next time a class or interface is compiled because there is no method or constructor that is most specific (§15.12.2.5), no such error occurs when a program is executed, because no overload resolution is done at execution time.

13.4.24 Method Overriding

If an instance method is added to a subclass and it overrides a method in a superclass, then the subclass method will be found by method invocations in pre-existing binaries, and these binaries are not impacted.

If a class method is added to a class, then this method will not be found unless the qualifying type of the reference is the subclass type.

13.4.25 Static Initializers

Adding, deleting, or changing a static initializer (§8.7) of a class does not impact pre-existing binaries.

13.4.26 Evolution of Enums

Adding or reordering constants in an enum type will not break compatibility with pre-existing binaries.

If a pre-existing binary attempts to access an enum constant that no longer exists, the client will fail at run-time with a `NoSuchFieldError`. Therefore such a change is not recommended for widely distributed enums.

In all other respects, the binary compatibility rules for enums are identical to those for classes.

13.5 Evolution of Interfaces

This section describes the impact of changes to the declaration of an interface and its members on pre-existing binaries.

13.5.1 `public` Interfaces

Changing an interface that is not declared `public` to be declared `public` does not break compatibility with pre-existing binaries.

If an interface that is declared `public` is changed to not be declared `public`, then an `IllegalAccessError` is thrown if a pre-existing binary is linked that needs but no longer has access to the interface type, so such a change is not recommended for widely distributed interfaces.

13.5.2 Superinterfaces

Changes to the interface hierarchy cause errors in the same way that changes to the class hierarchy do, as described in §13.4.4. In particular, changes that result in any previous superinterface of a class no longer being a superinterface can break compatibility with pre-existing binaries, resulting in a `VerifyError`.

13.5.3 The Interface Members

Adding a method to an interface does not break compatibility with pre-existing binaries.

A field added to a superinterface of `C` may hide a field inherited from a superclass of `C`. If the original reference was to an instance field, an `IncompatibleClassChangeError` will result. If the original reference was an assignment, an `IllegalAccessError` will result.

Deleting a member from an interface may cause linkage errors in pre-existing binaries.

13.5.4 Interface Type Parameters

The effects of changes to the type parameters of an interface are the same as those of analogous changes to the type parameters of a class.

13.5.5 Field Declarations

The considerations for changing field declarations in interfaces are the same as those for `static final` fields in classes, as described in §13.4.8 and §13.4.9.

13.5.6 abstract Methods

The considerations for changing abstract method declarations in interfaces are the same as those for abstract methods in classes, as described in §13.4.14, §13.4.15, §13.4.21, and §13.4.23.

13.5.7 Evolution of Annotation Types

Annotation types behave exactly like any other interface. Adding or removing an element from an annotation type is analogous to adding or removing a method. There are important considerations governing other changes to annotation types, but these have no effect on the linkage of binaries by the Java virtual machine. Rather, such changes affect the behavior of reflective APIs that manipulate annotations. The documentation of these APIs specifies their behavior when various changes are made to the underlying annotation types.

Adding or removing annotations has no effect on the correct linkage of the binary representations of programs in the Java programming language.

Blocks and Statements

THE sequence of execution of a program is controlled by *statements*, which are executed for their effect and do not have values.

Some statements *contain* other statements as part of their structure; such other statements are substatements of the statement. We say that statement *S* *immediately contains* statement *U* if there is no statement *T* different from *S* and *U* such that *S* contains *T* and *T* contains *U*. In the same manner, some statements contain expressions (Chapter 15, *Expressions*) as part of their structure.

The first section of this chapter discusses the distinction between normal and abrupt completion of statements (§14.1). Most of the remaining sections explain the various kinds of statements, describing in detail both their normal behavior and any special treatment of abrupt completion.

Blocks are explained first (§14.2), followed by local class declarations (§14.3) and local variable declaration statements (§14.4).

Next a grammatical maneuver that sidesteps the familiar "dangling else" problem (§14.5) is explained.

The last section (§14.21) of this chapter addresses the requirement that every statement be *reachable* in a certain technical sense.

14.1 Normal and Abrupt Completion of Statements

Every statement has a normal mode of execution in which certain computational steps are carried out. The following sections describe the normal mode of execution for each kind of statement.

If all the steps are carried out as described, with no indication of abrupt completion, the statement is said to *complete normally*. However, certain events may prevent a statement from completing normally:

- The `break` (§14.15), `continue` (§14.16), and `return` (§14.17) statements cause a transfer of control that may prevent normal completion of statements that contain them.
- Evaluation of certain expressions may throw exceptions from the Java virtual machine; these expressions are summarized in §15.6. An explicit `throw` (§14.18) statement also results in an exception. An exception causes a transfer of control that may prevent normal completion of statements.

If such an event occurs, then execution of one or more statements may be terminated before all steps of their normal mode of execution have completed; such statements are said to *complete abruptly*.

An abrupt completion always has an associated *reason*, which is one of the following:

- A `break` with no label
- A `break` with a given label
- A `continue` with no label
- A `continue` with a given label
- A `return` with no value
- A `return` with a given value
- A `throw` with a given value, including exceptions thrown by the Java virtual machine

The terms "complete normally" and "complete abruptly" also apply to the evaluation of expressions (§15.6). The only reason an expression can complete abruptly is that an exception is thrown, because of either a `throw` with a given value (§14.18) or a run-time exception or error (Chapter 11, *Exceptions*, §15.6).

If a statement evaluates an expression, abrupt completion of the expression always causes the immediate abrupt completion of the statement, with the same reason. All succeeding steps in the normal mode of execution are not performed.

Unless otherwise specified in this chapter, abrupt completion of a substatement causes the immediate abrupt completion of the statement itself, with the same reason, and all succeeding steps in the normal mode of execution of the statement are not performed.

Unless otherwise specified, a statement completes normally if all expressions it evaluates and all substatements it executes complete normally.

14.2 Blocks

A *block* is a sequence of statements, local class declarations, and local variable declaration statements within braces.

Block:

{ BlockStatements_{opt} }

BlockStatements:

BlockStatement

BlockStatements BlockStatement

BlockStatement:

LocalVariableDeclarationStatement

ClassDeclaration

Statement

A block is executed by executing each of the local variable declaration statements and other statements in order from first to last (left to right). If all of these block statements complete normally, then the block completes normally. If any of these block statements complete abruptly for any reason, then the block completes abruptly for the same reason.

14.3 Local Class Declarations

A *local class* is a nested class (Chapter 8, *Classes*) that is not a member of any class and that has a name.

All local classes are inner classes (§8.1.3).

Every local class declaration statement is immediately contained by a block. Local class declaration statements may be intermixed freely with other kinds of statements in the block.

The name of a local class *C* may not be redeclared as a local class of the directly enclosing method, constructor, or initializer block within the scope of *C*, or a compile-time error occurs.

It is a compile-time error if a local class declaration contains any one of the following access modifiers: `public`, `protected`, `private`, or `static`.

14.4 Local Variable Declaration Statements

A *local variable declaration statement* declares one or more local variable names.

LocalVariableDeclarationStatement:

LocalVariableDeclaration ;

LocalVariableDeclaration:

*VariableModifiers*_{opt} *Type* *VariableDeclarators*

Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

A local variable declaration can also appear in the header of a `for` statement (§14.14). In this case it is executed in the same manner as if it were part of a local variable declaration statement.

14.4.1 Local Variable Declarators and Types

Each *declarator* in a local variable declaration declares one local variable, whose name is the *Identifier* that appears in the declarator.

If the optional keyword `final` appears at the start of the declarator, the variable being declared is a final variable (§4.12.4).

If an annotation *a* on a local variable declaration corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.ElementType.LOCAL_VARIABLE`, or a compile-time error occurs. Annotation modifiers are described further in §9.7.

The declared type of a local variable is denoted by the *Type* that appears in the local variable declaration, followed by any bracket pairs that follow the *Identifier* in the declarator.

A local variable of type `float` always contains a value that is an element of the float value set (§4.2.3); similarly, a local variable of type `double` always contains a value that is an element of the double value set. It is not permitted for a local

variable of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a local variable of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

14.4.2 Local Variable Names

A local variable can only be referred to using a simple name (§6.5.6.1, not a qualified name).

The name of a local variable v may not be redeclared as a local variable of the directly enclosing method, constructor, or initializer block within the scope of v , or a compile-time error occurs.

The name of a local variable v may not be redeclared as an exception parameter of a catch clause in a try statement of the directly enclosing method, constructor or initializer block within the scope of v , or a compile-time error occurs.

If a declaration of an identifier as a local variable of a method, constructor, or initializer block appears within the scope of a parameter or local variable of the same name, a compile-time error occurs.

14.4.3 Execution of Local Variable Declarations

A local variable declaration statement is an executable statement. Every time it is executed, the declarators are processed in order from left to right. If a declarator has an initialization expression, the expression is evaluated and its value is assigned to the variable. If a declarator does not have an initialization expression, then a Java compiler must prove, using exactly the algorithm given in Chapter 16, *Definite Assignment*, that every reference to the variable is necessarily preceded by execution of an assignment to the variable. If this is not the case, then a compile-time error occurs.

Each initialization (except the first) is executed only if evaluation of the preceding initialization expression completes normally.

Execution of the local variable declaration completes normally only if evaluation of the last initialization expression completes normally.

If the local variable declaration contains no initialization expressions, then executing it always completes normally.

14.5 Statements

There are many kinds of statements in the Java programming language. Most correspond to statements in the C and C++ languages, but some are unique.

As in C and C++, the if statement of the Java programming language suffers from the so-called "dangling else problem," illustrated by this misleadingly formatted example:

```
if (door.isOpen())
    if (resident.isVisible())
        resident.greet("Hello!");
else door.bell.ring(); // A "dangling else"
```

The problem is that both the outer if statement and the inner if statement might conceivably own the else clause. In this example, one might surmise that the programmer intended the else clause to belong to the outer if statement.

The Java programming language, like C and C++ and many programming languages before them, arbitrarily decree that an else clause belongs to the innermost if to which it might possibly belong. This rule is captured by the following grammar:

```
Statement:
    StatementWithoutTrailingSubstatement
    LabeledStatement
    IfThenStatement
    IfThenElseStatement
    WhileStatement
    ForStatement
```

StatementWithoutTrailingSubstatement:

Block
EmptyStatement
ExpressionStatement
AssertStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

StatementNoShortIf:

StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

The following are repeated from §14.9 to make the presentation here clearer:

IfThenStatement:

if (Expression) Statement

IfThenElseStatement:

if (Expression) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:

if (Expression) StatementNoShortIf else StatementNoShortIf

Statements are thus grammatically divided into two categories: those that might end in an *if* statement that has no *else* clause (a "short *if* statement") and those that definitely do not.

Only statements that definitely do not end in a short *if* statement may appear as an immediate substatement before the keyword *else* in an *if* statement that does have an *else* clause.

This simple rule prevents the "dangling *else*" problem. The execution behavior of a statement with the "no short *if*" restriction is identical to the execution behavior

of the same kind of statement without the "no short if" restriction; the distinction is drawn purely to resolve the syntactic difficulty.

14.6 The Empty Statement

An empty statement does nothing.

EmptyStatement:
;

Execution of an empty statement always completes normally.

14.7 Labeled Statements

Statements may have *label* prefixes.

LabeledStatement:
Identifier : *Statement*

LabeledStatementNoShortIf:
Identifier : *StatementNoShortIf*

The *Identifier* is declared to be the label of the immediately contained *Statement*.

Unlike C and C++, the Java programming language has no `goto` statement; identifier statement labels are used with `break` (§14.15) or `continue` (§14.16) statements appearing anywhere within the labeled statement.

Let *l* be a label, and let *m* be the immediately enclosing method, constructor, instance initializer or static initializer. It is a compile-time error if *l* shadows (§6.4.1) the declaration of another label immediately enclosed in *m*.

There is no restriction against using the same identifier as a label and as the name of a package, class, interface, method, field, parameter, or local variable. Use of an identifier to label a statement does not obscure (§6.4.2) a package, class, interface, method, field, parameter, or local variable with the same name. Use of an identifier as a class, interface, method, field, local variable or as the parameter of an exception handler (§14.20) does not obscure a statement label with the same name.

A labeled statement is executed by executing the immediately contained *Statement*.

If the statement is labeled by an *Identifier* and the contained *Statement* completes abruptly because of a break with the same *Identifier*, then the labeled statement completes normally. In all other cases of abrupt completion of the *Statement*, the labeled statement completes abruptly for the same reason.

14.8 Expression Statements

Certain kinds of expressions may be used as statements by following them with semicolons:

ExpressionStatement:

StatementExpression ;

StatementExpression:

Assignment

PreIncrementExpression

PreDecrementExpression

PostIncrementExpression

PostDecrementExpression

MethodInvocation

ClassInstanceCreationExpression

An *expression statement* is executed by evaluating the expression; if the expression has a value, the value is discarded.

Execution of the expression statement completes normally if and only if evaluation of the expression completes normally.

Unlike C and C++, the Java programming language allows only certain forms of expressions to be used as expression statements. Note that the Java programming language does not allow a "cast to void" - void is not a type - so the traditional C trick of writing an expression statement such as:

```
(void)... ; // incorrect!
```

does not work. On the other hand, the language allows all the most useful kinds of expressions in expressions statements, and it does not require a method invocation used as an expression statement to invoke a void method, so such a trick is almost never needed. If a trick is needed, either an assignment statement (§15.26) or a local variable declaration statement (§14.4) can be used instead.

14.9 The if Statement

The *if* statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

IfThenStatement:

if (Expression) Statement

IfThenElseStatement:

if (Expression) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:

if (Expression) StatementNoShortIf else StatementNoShortIf

The *Expression* must have type *boolean* or *Boolean*, or a compile-time error occurs.

14.9.1 The if-then Statement

An *if-then* statement is executed by first evaluating the *Expression*. If the result is of type *Boolean*, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the *if-then* statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is *true*, then the contained *Statement* is executed; the *if-then* statement completes normally if and only if execution of the *Statement* completes normally.
- If the value is *false*, no further action is taken and the *if-then* statement completes normally.

14.9.2 The if-then-else Statement

An *if-then-else* statement is executed by first evaluating the *Expression*. If the result is of type *Boolean*, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, then the *if-then-else* statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the first contained *Statement* (the one before the `else` keyword) is executed; the `if-then-else` statement completes normally if and only if execution of that statement completes normally.
- If the value is `false`, then the second contained *Statement* (the one after the `else` keyword) is executed; the `if-then-else` statement completes normally if and only if execution of that statement completes normally.

14.10 The `assert` Statement

An *assertion* is an `assert` statement containing a boolean expression.

An assertion is either *enabled* or *disabled*. If the assertion is enabled, execution of the assertion causes evaluation of the boolean expression and an error is reported if the expression evaluates to `false`. If the assertion is disabled, execution of the assertion has no effect whatsoever.

AssertStatement:

```
assert Expression1 ;  
assert Expression1 : Expression2 ;
```

It is a compile-time error if *Expression1* does not have type `boolean` or `Boolean`.

In the second form of the `assert` statement, it is a compile-time error if *Expression2* is `void` (§15.1).

An `assert` statement that is executed after its class has completed initialization is enabled if and only if the host system has determined that the top level class that lexically contains the `assert` statement enables assertions.

Whether or not a top level class enables assertions is determined no later than the earliest of the initialization of the top level class and the initialization of any class nested in the top level class, and cannot be changed after it has been determined.

An `assert` statement that is executed before its class has completed initialization is enabled.

A disabled `assert` statement does nothing. In particular, neither *Expression1* nor *Expression2* (if it is present) are evaluated.

Execution of a disabled `assert` statement always completes normally.

An enabled `assert` statement is executed by first evaluating *Expression1*. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8).

If evaluation of *Expression1* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `assert` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the value of *Expression1*:

- If the value is `true`, no further action is taken and the `assert` statement completes normally.
- If the value is `false`, the execution behavior depends on whether *Expression2* is present:
 - ♦ If *Expression2* is present, it is evaluated.
 - ✦ If the evaluation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
 - ✦ If the evaluation completes normally, an `AssertionError` instance whose "detail message" is the resulting value of *Expression2* is created.
 - ♦ If the instance creation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
 - ♦ If the instance creation completes normally, the `assert` statement completes abruptly by throwing the newly created `AssertionError` object.
 - ♦ If *Expression2* is not present, an `AssertionError` instance with no "detail message" is created.
 - ✦ If the instance creation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
 - ✦ If the instance creation completes normally, the `assert` statement completes abruptly by throwing the newly created `AssertionError` object.

14.11 The switch Statement

The `switch` statement transfers control to one of several statements depending on the value of an expression.

SwitchStatement:

switch (Expression) SwitchBlock

SwitchBlock:

{ SwitchBlockStatementGroups_{opt} SwitchLabels_{opt} }

SwitchBlockStatementGroups:

SwitchBlockStatementGroup

SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:

SwitchLabels BlockStatements

SwitchLabels:

SwitchLabel

SwitchLabels SwitchLabel

SwitchLabel:

case ConstantExpression :

case EnumConstantName :

default :

EnumConstantName:

Identifier

The type of the *Expression* must be *char*, *byte*, *short*, *int*, *Character*, *Byte*, *Short*, *Integer*, or an enum type (§8.9), or a compile-time error occurs.

The body of a *switch* statement is known as a *switch block*. Any statement immediately contained by the *switch block* may be labeled with one or more *switch labels*, which are *case* or *default* labels. These labels are said to be *associated* with the *switch* statement, as are the values of the constant expressions (§15.28) or enum constants (§8.9.1) in the *case labels*.

All of the following must be true, or a compile-time error will result:

- Every case constant expression associated with a *switch* statement must be assignable (§5.2) to the type of the *switch Expression*.
- No *switch label* is *null*.
- No two of the case constant expressions associated with a *switch* statement may have the same value.

- At most one `default` label may be associated with the same `switch` statement.

When the `switch` statement is executed, first the *Expression* is evaluated. If the *Expression* evaluates to `null`, a `NullPointerException` is thrown and the entire `switch` statement completes abruptly for that reason. Otherwise, if the result is of a reference type, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `switch` statement completes abruptly for the same reason. Otherwise, execution continues by comparing the value of the *Expression* with each case constant, as follows:

- If one of the case constants is equal to the value of the expression, then we say that the case *matches*, and all statements after the matching case label in the `switch` block, if any, are executed in sequence.

If all these statements complete normally, or if there are no statements after the matching case label, then the entire `switch` statement completes normally.

- If no case matches but there is a `default` label, then all statements after the matching `default` label in the `switch` block, if any, are executed in sequence.

If all these statements complete normally, or if there are no statements after the `default` label, then the entire `switch` statement completes normally.

- If no case matches and there is no `default` label, then no further action is taken and the `switch` statement completes normally.

If any statement immediately contained by the *Block* body of the `switch` statement completes abruptly, it is handled as follows:

- If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `switch` statement completes normally.
- If execution of the *Statement* completes abruptly for any other reason, the `switch` statement completes abruptly for the same reason.

The case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

As in C and C++, execution of statements in a `switch` block "falls through labels."

14.12 The while Statement

The `while` statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is `false`.

WhileStatement:

```
while ( Expression ) Statement
```

WhileStatementNoShortIf:

```
while ( Expression ) StatementNoShortIf
```

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

A `while` statement is executed by first evaluating the *Expression*. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `while` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the contained *Statement* is executed. Then there is a choice:
 - ♦ If execution of the *Statement* completes normally, then the entire `while` statement is executed again, beginning by re-evaluating the *Expression*.
 - ♦ If execution of the *Statement* completes abruptly, see §14.12.1 below.
- If the (possibly unboxed) value of the *Expression* is `false`, no further action is taken and the `while` statement completes normally.

If the (possibly unboxed) value of the *Expression* is `false` the first time it is evaluated, then the *Statement* is not executed.

14.12.1 Abrupt Completion

Abrupt completion of the contained *Statement* is handled in the following manner:

- If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `while` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with no label, then the entire `while` statement is executed again.

- If execution of the *Statement* completes abruptly because of a `continue` with label *L*, then there is a choice:
 - ♦ If the `while` statement has label *L*, then the entire `while` statement is executed again.
 - ♦ If the `while` statement does not have label *L*, the `while` statement completes abruptly because of a `continue` with label *L*.
- If execution of the *Statement* completes abruptly for any other reason, the `while` statement completes abruptly for the same reason.

The case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

14.13 The do Statement

The `do` statement executes a *Statement* and an *Expression* repeatedly until the value of the *Expression* is `false`.

DoStatement:

```
do Statement while ( Expression ) ;
```

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

A `do` statement is executed by first executing the *Statement*. Then there is a choice:

- If execution of the *Statement* completes normally, then the *Expression* is evaluated. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `do` statement completes abruptly for the same reason. Otherwise, there is a choice based on the resulting value:

- ♦ If the value is `true`, then the entire `do` statement is executed again.
- ♦ If the value is `false`, no further action is taken and the `do` statement completes normally.
- If execution of the *Statement* completes abruptly, see §14.13.1 below.

Executing a `do` statement always executes the contained *Statement* at least once.

14.13.1 Abrupt Completion

Abrupt completion of the contained *Statement* is handled in the following manner:

- If execution of the *Statement* completes abruptly because of a `break` with no label, then no further action is taken and the `do` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with no label, then the *Expression* is evaluated. Then there is a choice based on the resulting value:
 - ♦ If the value is `true`, then the entire `do` statement is executed again.
 - ♦ If the value is `false`, no further action is taken and the `do` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with label *L*, then there is a choice:
 - ♦ If the `do` statement has label *L*, then the *Expression* is evaluated. Then there is a choice:
 - ✧ If the value of the *Expression* is `true`, then the entire `do` statement is executed again.
 - ✧ If the value of the *Expression* is `false`, no further action is taken and the `do` statement completes normally.
 - ♦ If the `do` statement does not have label *L*, the `do` statement completes abruptly because of a `continue` with label *L*.
- If execution of the *Statement* completes abruptly for any other reason, the `do` statement completes abruptly for the same reason.

The case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

14.14 The `for` Statement

The `for` statement has two forms:

- The basic `for` statement.
- The enhanced `for` statement

ForStatement:

BasicForStatement

EnhancedForStatement

14.14.1 The basic `for` Statement

The basic `for` statement executes some initialization code, then executes an *Expression*, a *Statement*, and some update code repeatedly until the value of the *Expression* is false.

BasicForStatement:

for (*ForInit_{opt}* ; *Expression_{opt}* ; *ForUpdate_{opt}*) *Statement*

ForStatementNoShortIf:

for (*ForInit_{opt}* ; *Expression_{opt}* ; *ForUpdate_{opt}*) *StatementNoShortIf*

ForInit:

StatementExpressionList

LocalVariableDeclaration

ForUpdate:

StatementExpressionList

StatementExpressionList:

StatementExpression

StatementExpressionList , *StatementExpression*

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

14.14.1.1 Initialization of `for` statement

A `for` statement is executed by first executing the *ForInit* code:

- If the *ForInit* code is a list of statement expressions (§14.8), the expressions are evaluated in sequence from left to right; their values, if any, are discarded.

If evaluation of any expression completes abruptly for some reason, the `for` statement completes abruptly for the same reason; any *ForInit* statement expressions to the right of the one that completed abruptly are not evaluated.

- If the *ForInit* code is a local variable declaration, it is executed as if it were a local variable declaration statement (§14.4) appearing in a block.

If execution of the local variable declaration completes abruptly for any reason, the *for* statement completes abruptly for the same reason.

- If the *ForInit* part is not present, no action is taken.

14.14.1.2 Iteration of *for* statement

Next, a *for* iteration step is performed, as follows:

- If the *Expression* is present, it is evaluated. If the result is of type *Boolean*, it is subject to unboxing conversion (§5.1.8).

If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly, the *for* statement completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present; see next bullet.

- If the *Expression* is not present, or it is present and the value resulting from its evaluation (including any possible unboxing) is *true*, then the contained *Statement* is executed. Then there is a choice:

- ♦ If execution of the *Statement* completes normally, then the following two steps are performed in sequence:

1. First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the *for* statement completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated.

If the *ForUpdate* part is not present, no action is taken.

2. Second, another *for* iteration step is performed.

- ♦ If execution of the *Statement* completes abruptly, see §14.14.1.3 below.
- If the *Expression* is present and the value resulting from its evaluation (including any possible unboxing) is *false*, no further action is taken and the *for* statement completes normally.

If the (possibly unboxed) value of the *Expression* is *false* the first time it is evaluated, then the *Statement* is not executed.

If the *Expression* is not present, then the only way a *for* statement can complete normally is by use of a *break* statement.

14.14.1.3 Abrupt Completion of for statement

Abrupt completion of the contained *Statement* is handled in the following manner:

- If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `for` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with no label, then the following two steps are performed in sequence:
 1. First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If the *ForUpdate* part is not present, no action is taken.
 2. Second, another `for` iteration step is performed.
- If execution of the *Statement* completes abruptly because of a `continue` with label *L*, then there is a choice:
 - ♦ If the `for` statement has label *L*, then the following two steps are performed in sequence:
 1. First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If the *ForUpdate* is not present, no action is taken.
 2. Second, another `for` iteration step is performed.
 - ♦ If the `for` statement does not have label *L*, the `for` statement completes abruptly because of a `continue` with label *L*.
- If execution of the *Statement* completes abruptly for any other reason, the `for` statement completes abruptly for the same reason.

The case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

14.14.2 The enhanced for statement

The enhanced `for` statement has the form:

EnhancedForStatement:

`for ( FormalParameter : Expression ) Statement`

The *Expression* must either have type `Iterable` or else it must be of an array type (§10.1), or a compile-time error occurs.

The meaning of the enhanced `for` statement is given by translation into a basic `for` statement, as follows:

- If the type of *Expression* is a subtype of `Iterable`, then let *I* be the type of the expression *Expression*.`iterator()`.

The enhanced `for` statement is equivalent to a basic `for` statement of the form:

```
for (I i = Expression.iterator(); i.hasNext(); ) {
    VariableModifiersopt Type Identifier = (TargetType) i.next();
    Statement
}
```

i is an automatically generated identifier that is distinct from any other identifiers (automatically generated or otherwise) that are in scope (§6.3) at the point where the enhanced `for` statement occurs.

If *Type* is a reference type, then *TargetType* is *Type*; otherwise, *TargetType* is the upper bound of the capture conversion of the type argument of *I*, or `Object` if *I* is raw.

- Otherwise, the *Expression* necessarily has an array type, *T*[],. Let *L*₁ ... *L*_{*m*} be the (possibly empty) sequence of labels immediately preceding the enhanced `for` statement.

The enhanced `for` statement is equivalent to a basic `for` statement of the form:

```
T[] a = Expression;
L1: L2: ... Lm:
for (int i = 0; i < a.length; i++) {
    VariableModifiersopt TargetType Identifier = a[i];
    Statement
}
```

a and *i* are automatically generated identifiers that are distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the enhanced `for` statement occurs.

TargetType is the type of the loop variable as denoted by the *Type* that appears in the *FormalParameter*, followed by any bracket pairs that follow the *Identifier* in the *FormalParameter*.

14.15 The `break` Statement

A `break` statement transfers control out of an enclosing statement.

BreakStatement:

```
break Identifieropt ;
```

A break statement with no label attempts to transfer control to the innermost enclosing switch, while, do, or for statement of the immediately enclosing method or initializer block; this statement, which is called the *break target*, then immediately completes normally.

To be precise, a break statement with no label always completes abruptly, the reason being a break with no label.

If no switch, while, do, or for statement in the immediately enclosing method, constructor, or initializer encloses the break statement, a compile-time error occurs.

A break statement with label *Identifier* attempts to transfer control to the enclosing labeled statement (§14.7) that has the same *Identifier* as its label; this statement, which is called the *break target*, then immediately completes normally. In this case, the break target need not be a switch, while, do, or for statement.

A break statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

If no labeled statement with *Identifier* as its label encloses the break statement, a compile-time error occurs.

To be precise, a break statement with label *Identifier* always completes abruptly, the reason being a break with label *Identifier*.

It can be seen, then, that a break statement always completes abruptly.

14.16 The continue Statement

A continue statement may occur only in a while, do, or for statement; statements of these three kinds are called *iteration statements*. Control passes to the loop-continuation point of an iteration statement.

ContinueStatement:

```
continue Identifieropt ;
```

A continue statement with no label attempts to transfer control to the innermost enclosing while, do, or for statement of the immediately enclosing method or initializer block; this statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one.

To be precise, such a `continue` statement always completes abruptly, the reason being a `continue` with no label.

If no `while`, `do`, or `for` statement of the immediately enclosing method or initializer block encloses the `continue` statement, a compile-time error occurs.

A `continue` statement with label *Identifier* attempts to transfer control to the enclosing labeled statement (§14.7) that has the same *Identifier* as its label; that statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one.

The `continue target` must be a `while`, `do`, or `for` statement, or a compile-time error occurs.

A `continue` statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

If no labeled statement with *Identifier* as its label contains the `continue` statement, a compile-time error occurs.

To be precise, a `continue` statement with label *Identifier* always completes abruptly, the reason being a `continue` with label *Identifier*.

It can be seen, then, that a `continue` statement always completes abruptly.

14.17 The `return` Statement

A `return` statement returns control to the invoker of a method (§8.4, §15.12) or constructor (§8.8, §15.9).

ReturnStatement:
`return Expressionopt ;`

A `return` statement is *contained* in the innermost constructor, method, or initializer whose body encloses the `return` statement.

It is a compile-time error if a `return` statement is contained in an instance initializer (§8.6) or a static initializer (§8.7).

A `return` statement with no *Expression* must be contained in a method that is declared, using the keyword `void`, not to return any value (§8.4), or in a constructor (§8.8), or a compile-time error occurs.

A return statement with no *Expression* attempts to transfer control to the invoker of the method or constructor that contains it. To be precise, a return statement with no *Expression* always completes abruptly, the reason being a return with no value.

A return statement with an *Expression* must be contained in a method declaration that is declared to return a value (§8.4), or a compile-time error occurs.

The *Expression* must denote a variable or value of some type *T*, or a compile-time error occurs.

The type *T* must be assignable (§5.2) to the declared result type of the method, or a compile-time error occurs.

A return statement with an *Expression* attempts to transfer control to the invoker of the method that contains it; the value of the *Expression* becomes the value of the method invocation. More precisely, execution of such a return statement first evaluates the *Expression*. If the evaluation of the *Expression* completes abruptly for some reason, then the return statement completes abruptly for that reason. If evaluation of the *Expression* completes normally, producing a value *V*, then the return statement completes abruptly, the reason being a return with value *V*.

If the expression is of type `float` and is not FP-strict (§15.4), then the value may be an element of either the float value set or the float-extended-exponent value set (§4.2.3). If the expression is of type `double` and is not FP-strict, then the value may be an element of either the double value set or the double-extended-exponent value set.

It can be seen, then, that a return statement always completes abruptly.

14.18 The `throw` Statement

A `throw` statement causes an exception (Chapter 11, *Exceptions*) to be thrown. The result is an immediate transfer of control (§11.3) that may exit multiple statements and multiple constructor, instance initializer, static initializer and field initializer evaluations, and method invocations until a `try` statement (§14.20) is found that catches the thrown value. If no such `try` statement is found, then execution of the thread (Chapter 17, *Threads and Locks*) that executed the `throw` is terminated (§11.3) after invocation of the `uncaughtException` method for the thread group to which the thread belongs.

ThrowStatement:
`throw Expression ;`

The *Expression* in a `throw` statement must denote either 1) a variable or value of a reference type which is assignable (§5.2) to the type `Throwable`, or 2) the null reference, or a compile-time error occurs.

At least one of the following three conditions must be true, or a compile-time error occurs:

- The type of the *Expression* is an unchecked exception class (§11.1.1).
- The `throw` statement is contained in the `try` block of a `try` statement (§14.20) and it is not the case that the `try` statement can throw an exception of the type of the *Expression*. (In this case we say the thrown value is *caught* by the `try` statement.)
- The `throw` statement is contained in a method or constructor declaration and the type of the *Expression* is assignable (§5.2) to at least one type listed in the `throws` clause (§8.4.6, §8.8.5) of the declaration.

A `throw` statement first evaluates the *Expression*. Then:

- If evaluation of the *Expression* completes abruptly for some reason, then the `throw` completes abruptly for that reason.
- If evaluation of the *Expression* completes normally, producing a non-`null` value *V*, then the `throw` statement completes abruptly, the reason being a `throw` with value *V*.
- If evaluation of the *Expression* completes normally, producing a `null` value, then an instance *V'* of class `NullPointerException` is created and thrown instead of `null`. The `throw` statement then completes abruptly, the reason being a `throw` with value *V'*.

If there are any enclosing `try` statements (§14.20) whose `try` blocks contain the `throw` statement, then any `finally` clauses of those `try` statements are executed as control is transferred outward, until the thrown value is caught. Note that abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `throw` statement.

If a `throw` statement is contained in a method declaration, but its value is not caught by some `try` statement that contains it, then the invocation of the method completes abruptly because of the `throw`.

If a `throw` statement is contained in a constructor declaration, but its value is not caught by some `try` statement that contains it, then the class instance creation expression that invoked the constructor will complete abruptly because of the `throw`.

If a `throw` statement is contained in a static initializer (§8.7), then a compile-time check (§11.2.3) ensures that either its value is always an unchecked exception or its value is always caught by some `try` statement that contains it. If at run-time, despite this check, the value is not caught by some `try` statement that contains the `throw` statement, then the value is rethrown if it is an instance of class `Error` or one of its subclasses; otherwise, it is wrapped in an `ExceptionInInitializerError` object, which is then thrown (§12.4.2).

If a `throw` statement is contained in an instance initializer (§8.6), then a compile-time check (§11.2.3) ensures that either its value is always an unchecked exception or its value is always caught by some `try` statement that contains it, or the type of the thrown exception (or one of its superclasses) occurs in the `throws` clause of every constructor of the class.

14.19 The synchronized Statement

A `synchronized` statement acquires a mutual-exclusion lock (§17.1) on behalf of the executing thread, executes a block, then releases the lock. While the executing thread owns the lock, no other thread may acquire the lock.

SynchronizedStatement:

`synchronized ( Expression ) Block`

The type of *Expression* must be a reference type, or a compile-time error occurs.

A `synchronized` statement is executed by first evaluating the *Expression*. Then:

- If evaluation of the *Expression* completes abruptly for some reason, then the `synchronized` statement completes abruptly for the same reason.
- Otherwise, if the value of the *Expression* is `null`, a `NullPointerException` is thrown.
- Otherwise, let the non-`null` value of the *Expression* be *v*. The executing thread locks the lock associated with *v*. Then the *Block* is executed. If execution of the *Block* completes normally, then the lock is unlocked and the `synchronized` statement completes normally. If execution of the *Block* completes abruptly for any reason, then the lock is unlocked and the `synchronized` statement completes abruptly for the same reason.

The locks acquired by `synchronized` statements are the same as the locks that are acquired implicitly by `synchronized` methods (§8.4.3.6). A single thread may hold a lock more than once.

Acquiring the lock associated with an object does not in itself prevent other threads from accessing fields of the object or invoking un-synchronized methods on the object. Other threads can also use synchronized methods or the synchronized statement in a conventional manner to achieve mutual exclusion.

14.20 The `try` statement

A `try` statement executes a block. If a value is thrown and the `try` statement has one or more catch clauses that can catch it, then control will be transferred to the first such catch clause. If the `try` statement has a `finally` clause, then another block of code is executed, no matter whether the `try` block completes normally or abruptly, and no matter whether a catch clause is first given control.

TryStatement:

```
try Block Catches
try Block Catchesopt Finally
```

Catches:

```
CatchClause
Catches CatchClause
```

CatchClause:

```
catch ( FormalParameter ) Block
```

Finally:

```
finally Block
```

The *Block* immediately after the keyword `try` is called the *try block* of the `try` statement.

The *Block* immediately after the keyword `finally` is called the *finally block* of the `try` statement.

A `try` statement may have catch clauses, also called *exception handlers*.

A catch clause must have exactly one parameter, which is called an *exception parameter*.

The declared type of the exception parameter must be the class `Throwable` or a subclass (not just a subtype) of `Throwable`, or a compile-time error occurs.

It is a compile-time error if an exception parameter that is declared `final` is assigned to within the body of the catch clause.

An exception parameter can only be referred to by a simple name (§6.5.6.1), not a qualified name.

An exception parameter of a catch clause must not have the same name as a local variable or parameter of the method or initializer block immediately enclosing the catch clause, or a compile-time error occurs.

Within the *Block* of the catch clause, the name of the parameter may not be redeclared as a local variable of the directly enclosing method or initializer block, nor may it be redeclared as an exception parameter of a catch clause in a try statement of the directly enclosing method or initializer block, or a compile-time error occurs.

Exception handlers are considered in left-to-right order: the earliest possible catch clause accepts the exception, receiving as its actual argument the thrown exception object.

A `finally` clause ensures that the `finally` block is executed after the `try` block and any catch block that might be executed, no matter how control leaves the `try` block or catch block.

Handling of the `finally` block is rather complex, so the two cases of a try statement with and without a `finally` block are described separately.

14.20.1 Execution of try-catch

A try statement without a `finally` block is executed by first executing the `try` block. Then there is a choice:

- If execution of the `try` block completes normally, then no further action is taken and the try statement completes normally.
- If execution of the `try` block completes abruptly because of a throw of a value *V*, then there is a choice:
 - ♦ If the run-time type of *V* is assignable (§5.2) to the *Parameter* of any catch clause of the try statement, then the first (leftmost) such catch clause is selected. The value *V* is assigned to the parameter of the selected catch clause, and the *Block* of that catch clause is executed.

If that block completes normally, then the try statement completes normally; if that block completes abruptly for any reason, then the try statement completes abruptly for the same reason.

- ♦ If the run-time type of V is not assignable to the parameter of any catch clause of the try statement, then the try statement completes abruptly because of a throw of the value V .
- If execution of the try block completes abruptly for any other reason, then the try statement completes abruptly for the same reason.

14.20.2 Execution of try-finally and try-catch-finally

A try statement with a finally block is executed by first executing the try block. Then there is a choice:

- If execution of the try block completes normally, then the finally block is executed, and then there is a choice:
 - ♦ If the finally block completes normally, then the try statement completes normally.
 - ♦ If the finally block completes abruptly for reason S , then the try statement completes abruptly for reason S .
- If execution of the try block completes abruptly because of a throw of a value V , then there is a choice:
 - ♦ If the run-time type of V is assignable to the parameter of any catch clause of the try statement, then the first (leftmost) such catch clause is selected. The value V is assigned to the parameter of the selected catch clause, and the *Block* of that catch clause is executed. Then there is a choice:
 - ❖ If the catch block completes normally, then the finally block is executed. Then there is a choice:
 - ♦ If the finally block completes normally, then the try statement completes normally.
 - ♦ If the finally block completes abruptly for any reason, then the try statement completes abruptly for the same reason.
 - ❖ If the catch block completes abruptly for reason R , then the finally block is executed. Then there is a choice:
 - ♦ If the finally block completes normally, then the try statement completes abruptly for reason R .
 - ♦ If the finally block completes abruptly for reason S , then the try statement completes abruptly for reason S (and reason R is discarded).

- ♦ If the run-time type of V is not assignable to the parameter of any catch clause of the try statement, then the finally block is executed. Then there is a choice:
 - ♦ If the finally block completes normally, then the try statement completes abruptly because of a throw of the value V .
 - ♦ If the finally block completes abruptly for reason S , then the try statement completes abruptly for reason S (and the throw of value V is discarded and forgotten).
- If execution of the try block completes abruptly for any other reason R , then the finally block is executed. Then there is a choice:
 - ♦ If the finally block completes normally, then the try statement completes abruptly for reason R .
 - ♦ If the finally block completes abruptly for reason S , then the try statement completes abruptly for reason S (and reason R is discarded).

14.21 Unreachable Statements

It is a compile-time error if a statement cannot be executed because it is *unreachable*.

The rules in this section define two technical terms:

- whether a statement is *reachable*
- whether a statement *can complete normally*

The definitions here allow a statement to complete normally only if it is reachable.

A reachable break statement *exits a statement* if, within the break target, either there are no try statements whose try blocks contain the break statement, or there are try statements whose try blocks contain the break statement and all finally clauses of those try statements can complete normally.

A continue statement *continues a do statement* if, within the do statement, either there are no try statements whose try blocks contain the continue statement, or there are try statements whose try blocks contain the continue statement and all finally clauses of those try statements can complete normally.

The rules are as follows:

- The block that is the body of a constructor, method, instance initializer, or static initializer is reachable.
- An empty block that is not a switch block can complete normally iff it is reachable.

A non-empty block that is not a switch block can complete normally iff the last statement in it can complete normally.

The first statement in a non-empty block that is not a switch block is reachable iff the block is reachable.

Every other statement *S* in a non-empty block that is not a switch block is reachable iff the statement preceding *S* can complete normally.

- A local class declaration statement can complete normally iff it is reachable.
- A local variable declaration statement can complete normally iff it is reachable.
- An empty statement can complete normally iff it is reachable.
- A labeled statement can complete normally if at least one of the following is true:

- ♦ The contained statement can complete normally.
- ♦ There is a reachable `break` statement that exits the labeled statement.

The contained statement is reachable iff the labeled statement is reachable.

- An expression statement can complete normally iff it is reachable.
- The `if` statement, whether or not it has an `else` part, is handled in an unusual manner. For this reason, it is discussed separately at the end of this section.
- An `assert` statement can complete normally iff it is reachable.
- A `switch` statement can complete normally iff at least one of the following is true:

- ♦ The switch block is empty or contains only switch labels.
- ♦ The last statement in the switch block can complete normally.
- ♦ There is at least one switch label after the last switch block statement group.
- ♦ The switch block does not contain a `default` label.
- ♦ There is a reachable `break` statement that exits the switch statement.

- A switch block is reachable iff its `switch` statement is reachable.

- A statement in a switch block is reachable iff its `switch` statement is reachable and at least one of the following is true:
 - ♦ It bears a `case` or `default` label.
 - ♦ There is a statement preceding it in the switch block and that preceding statement can complete normally.
- A `while` statement can complete normally iff at least one of the following is true:
 - ♦ The `while` statement is reachable and the condition expression is not a constant expression with value `true`.
 - ♦ There is a reachable `break` statement that exits the `while` statement.

The contained statement is reachable iff the `while` statement is reachable and the condition expression is not a constant expression whose value is `false`.

- A `do` statement can complete normally iff at least one of the following is true:
 - ♦ The contained statement can complete normally and the condition expression is not a constant expression with value `true`.
 - ♦ The `do` statement contains a reachable `continue` statement with no label, and the `do` statement is the innermost `while`, `do`, or `for` statement that contains that `continue` statement, and the `continue` statement continues that `do` statement, and the condition expression is not a constant expression with value `true`.
 - ♦ The `do` statement contains a reachable `continue` statement with a label *L*, and the `do` statement has label *L*, and the `continue` statement continues that `do` statement, and the condition expression is not a constant expression with value `true`.
 - ♦ There is a reachable `break` statement that exits the `do` statement.

The contained statement is reachable iff the `do` statement is reachable.

- A basic `for` statement can complete normally iff at least one of the following is true:
 - ♦ The `for` statement is reachable, there is a condition expression, and the condition expression is not a constant expression with value `true`.
 - ♦ There is a reachable `break` statement that exits the `for` statement.

The contained statement is reachable iff the `for` statement is reachable and the condition expression is not a constant expression whose value is `false`.

- An enhanced `for` statement can complete normally iff it is reachable.

- A `break`, `continue`, `return`, or `throw` statement cannot complete normally.
- A `synchronized` statement can complete normally iff the contained statement can complete normally.

The contained statement is reachable iff the `synchronized` statement is reachable.

- A `try` statement can complete normally iff both of the following are true:
 - ♦ The `try` block can complete normally or any `catch` block can complete normally.
 - ♦ If the `try` statement has a `finally` block, then the `finally` block can complete normally.
- The `try` block is reachable iff the `try` statement is reachable.
- A `catch` block *C* is reachable iff both of the following are true:

- ♦ Either the type of *C*'s parameter is an unchecked exception type or `Throwable`; or some expression or `throw` statement in the `try` block is reachable and can throw a checked exception whose type is assignable to the parameter of the catch clause *C*.

An expression is reachable iff the innermost statement containing it is reachable.

- ♦ There is no earlier `catch` block *A* in the `try` statement such that the type of *C*'s parameter is the same as or a subclass of the type of *A*'s parameter.
- The *Block* of a `catch` block is reachable iff the `catch` block is reachable.
- If a `finally` block is present, it is reachable iff the `try` statement is reachable.

The rules for the `if` statement are as follows:

- An `if-then` statement can complete normally iff it is reachable.

The `then`-statement is reachable iff the `if-then` statement is reachable.

- An `if-then-else` statement can complete normally iff the `then`-statement can complete normally or the `else`-statement can complete normally.

The `then`-statement is reachable iff the `if-then-else` statement is reachable.

The `else`-statement is reachable iff the `if-then-else` statement is reachable.

Expressions

MUCH of the work in a program is done by evaluating *expressions*, either for their side effects, such as assignments to variables, or for their values, which can be used as arguments or operands in larger expressions, or to affect the execution sequence in statements, or both.

This chapter specifies the meanings of expressions and the rules for their evaluation.

15.1 Evaluation, Denotation, and Result

When an expression in a program is *evaluated* (*executed*), the result denotes one of three things:

- A variable (§4.12) (in C, this would be called an *lvalue*)
- A value (§4.2, §4.3)
- Nothing (the expression is said to be void)

Evaluation of an expression can also produce side effects, because expressions may contain embedded assignments, increment operators, decrement operators, and method invocations.

An expression denotes nothing if and only if it is a method invocation (§15.12) that invokes a method that does not return a value, that is, a method declared `void` (§8.4). Such an expression can be used only as an expression statement (§14.8), because every other context in which an expression can appear requires the expression to denote something. An expression statement that is a method invocation may also invoke a method that produces a result; in this case the value returned by the method is quietly discarded.

Value set conversion (§5.1.13) is applied to the result of every expression that produces a value.

Each expression occurs in either:

- The declaration of some (class or interface) type that is being declared: in a field initializer, in a static initializer, in an instance initializer, in a constructor declaration, in an annotation, or in the code for a method.
- An annotation of a package or of a top-level type declaration.

15.2 Variables as Values

If an expression denotes a variable, and a value is required for use in further evaluation, then the value of that variable is used. In this context, if the expression denotes a variable or a value, we may speak simply of the *value* of the expression.

If the value of a variable of type `float` or `double` is used in this manner, then value set conversion (§5.1.13) is applied to the value of the variable.

15.3 Type of an Expression

If an expression denotes a variable or a value, then the expression has a type known at compile time. The rules for determining the type of an expression are explained separately below for each kind of expression.

The value of an expression is assignment compatible (§5.2) with the type of the expression, unless heap pollution (§4.12.2) occurs.

Likewise, the value stored in a variable is always compatible with the type of the variable, unless heap pollution occurs.

In other words, the value of an expression whose type is T is always suitable for assignment to a variable of type T .

Note that an expression whose type is a class type F that is declared `final` is guaranteed to have a value that is either a null reference or an object whose class is F itself, because `final` types have no subclasses.

15.4 FP-strict Expressions

If the type of an expression is `float` or `double`, then there is a question as to what value set (§4.2.3) the value of the expression is drawn from. This is governed by the rules of value set conversion (§5.1.13); these rules in turn depend on whether or not the expression is *FP-strict*.

Every compile-time constant expression (§15.28) is FP-strict.

If an expression is not a compile-time constant expression, then consider all the class declarations, interface declarations, and method declarations that contain the expression. If *any* such declaration bears the `strictfp` modifier, then the expression is FP-strict.

If a class, interface, or method, *X*, is declared `strictfp`, then *X* and any class, interface, method, constructor, instance initializer, static initializer or variable initializer within *X* is said to be *FP-strict*.

It follows that an expression is not FP-strict if and only if it is not a compile-time constant expression *and* it does not appear within any declaration that has the `strictfp` modifier.

Within an FP-strict expression, all intermediate values must be elements of the float value set or the double value set, implying that the results of all FP-strict expressions must be those predicted by IEEE 754 arithmetic on operands represented using single and double formats.

Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results; the net effect, roughly speaking, is that a calculation might produce "the correct answer" in situations where exclusive use of the float value set or double value set might result in overflow or underflow.

15.5 Expressions and Run-Time Checks

If the type of an expression is a primitive type, then the value of the expression is of that same primitive type.

If the type of an expression is a reference type, then the class of the referenced object, or even whether the value is a reference to an object rather than `null`, is not necessarily known at compile time. There are a few places in the Java programming language where the actual class of a referenced object affects program execution

in a manner that cannot be deduced from the type of the expression. They are as follows:

- Method invocation (§15.12). The particular method used for an invocation `o.m(...)` is chosen based on the methods that are part of the class or interface that is the type of `o`. For instance methods, the class of the object referenced by the run-time value of `o` participates because a subclass may override a specific method already declared in a parent class so that this overriding method is invoked. (The overriding method may or may not choose to further invoke the original overridden `m` method.)
- The `instanceof` operator (§15.20.2). An expression whose type is a reference type may be tested using `instanceof` to find out whether the class of the object referenced by the run-time value of the expression is assignment compatible (§5.2) with some other reference type.
- Casting (§5.5, §15.16). The class of the object referenced by the run-time value of the operand expression might not be compatible with the type specified by the cast. For reference types, this may require a run-time check that throws an exception if the class of the referenced object, as determined at run time, is not assignment compatible (§5.2) with the target type.
- Assignment to an array component of reference type (§10.5, §15.13, §15.26.1). The type-checking rules allow the array type `S[]` to be treated as a subtype of `T[]` if `S` is a subtype of `T`, but this requires a run-time check for assignment to an array component, similar to the check performed for a cast.
- Exception handling (§14.20). An exception is caught by a `catch` clause only if the class of the thrown exception object is an `instanceof` the type of the formal parameter of the `catch` clause.

Situations where the class of an object is not statically known may lead to run-time type errors.

In addition, there are situations where the statically known type may not be accurate at run-time. Such situations can arise in a program that gives rise to unchecked warnings. Such warnings are given in response to operations that cannot be statically guaranteed to be safe, and cannot immediately be subjected to dynamic checking because they involve non-reifiable (§4.7) types. As a result, dynamic checks later in the course of program execution may detect inconsistencies and result in run-time type errors.

A run-time type error can occur only in these situations:

- In a cast, when the actual class of the object referenced by the value of the operand expression is not compatible with the target type specified by the cast operator (§5.5, §15.16); in this case a `ClassCastException` is thrown.
- In an automatically generated cast introduced to ensure the validity of an operation on a non-reifiable type (§4.7).
- In an assignment to an array component of reference type, when the actual class of the object referenced by the value to be assigned is not compatible with the actual run-time component type of the array (§10.5, §15.13, §15.26.1); in this case an `ArrayStoreException` is thrown.
- When an exception is not caught by any catch clause of a try statement (§14.20); in this case the thread of control that encountered the exception first attempts to invoke an uncaught exception handler (§11.3) and then terminates.

15.6 Normal and Abrupt Completion of Evaluation

Every expression has a normal mode of evaluation in which certain computational steps are carried out. The following sections describe the normal mode of evaluation for each kind of expression.

If all the steps are carried out without an exception being thrown, the expression is said to *complete normally*. If, however, evaluation of an expression throws an exception, then the expression is said to *complete abruptly*. An abrupt completion always has an associated reason, which is always a throw with a given value.

Run-time exceptions are thrown by the predefined operators as follows:

- A class instance creation expression (§15.9), array creation expression (§15.10), array initializer expression (§10.6), or string concatenation operator expression (§15.18.1) throws an `OutOfMemoryError` if there is insufficient memory available.
- An array creation expression (§15.10) throws a `NegativeArraySizeException` if the value of any dimension expression is less than zero.
- A field access expression (§15.11) throws a `NullPointerException` if the value of the object reference expression is `null`.
- A method invocation expression (§15.12) that invokes an instance method throws a `NullPointerException` if the target reference is `null`.

- An array access expression (§15.13) throws a `NullPointerException` if the value of the array reference expression is `null`.
- An array access expression (§15.13) throws an `ArrayIndexOutOfBoundsException` if the value of the array index expression is negative or greater than or equal to the length of the array.
- A cast expression (§15.16) throws a `ClassCastException` if a cast is found to be impermissible at run time.
- An integer division (§15.17.2) or integer remainder (§15.17.3) operator throws an `ArithmeticException` if the value of the right-hand operand expression is zero.
- An assignment to an array component of reference type (§15.26.1), a method invocation expression (§15.12), or a prefix or postfix increment (§15.14.2, §15.15.1) or decrement operator (§15.14.3, §15.15.2) may all throw an `OutOfMemoryError` as a result of boxing conversion (§5.1.7).
- An assignment to an array component of reference type (§15.26.1) throws an `ArrayStoreException` when the value to be assigned is not compatible with the component type of the array (§10.5).

A method invocation expression can also result in an exception being thrown if an exception occurs that causes execution of the method body to complete abruptly. A class instance creation expression can also result in an exception being thrown if an exception occurs that causes execution of the constructor to complete abruptly. Various linkage and virtual machine errors may also occur during the evaluation of an expression. By their nature, such errors are difficult to predict and difficult to handle.

If an exception occurs, then evaluation of one or more expressions may be terminated before all steps of their normal mode of evaluation are complete; such expressions are said to complete abruptly.

The terms "complete normally" and "complete abruptly" are also applied to the execution of statements (§14.1). A statement may complete abruptly for a variety of reasons, not just because an exception is thrown.

If evaluation of an expression requires evaluation of a subexpression, then abrupt completion of the subexpression always causes the immediate abrupt completion of the expression itself, with the same reason, and all succeeding steps in the normal mode of evaluation are not performed.

15.7 Evaluation Order

The Java programming language guarantees that the operands of operators appear to be evaluated in a specific *evaluation order*, namely, from left to right.

It is recommended that code not rely crucially on this specification. Code is usually clearer when each expression contains at most one side effect, as its outermost operation, and when code does not depend on exactly which exception arises as a consequence of the left-to-right evaluation of expressions.

15.7.1 Evaluate Left-Hand Operand First

The left-hand operand of a binary operator appears to be fully evaluated before any part of the right-hand operand is evaluated.

If the operator is a compound-assignment operator (§15.26.2), then evaluation of the left-hand operand includes both remembering the variable that the left-hand operand denotes and fetching and saving that variable's value for use in the implied combining operation.

If evaluation of the left-hand operand of a binary operator completes abruptly, no part of the right-hand operand appears to have been evaluated.

15.7.2 Evaluate Operands before Operation

The Java programming language guarantees that every operand of an operator (except the conditional operators `&&`, `||`, and `? :`) appears to be fully evaluated before any part of the operation itself is performed.

If the binary operator is an integer division `/` (§15.17.2) or integer remainder `%` (§15.17.3), then its execution may raise an `ArithmeticException`, but this exception is thrown only after both operands of the binary operator have been evaluated and only if these evaluations completed normally.

15.7.3 Evaluation Respects Parentheses and Precedence

The Java programming language respects the order of evaluation indicated explicitly by parentheses and implicitly by operator precedence.

In the case of floating-point calculations, this rule applies also for infinity and not-a-number (NaN) values.

Specifically, floating-point calculations that appear to be mathematically associative are unlikely to be computationally associative. Such computations must not be naively reordered.

In contrast, integer addition and multiplication *are* provably associative in the Java programming language.

15.7.4 Argument Lists are Evaluated Left-to-Right

In a method or constructor invocation or class instance creation expression, argument expressions may appear within the parentheses, separated by commas. Each argument expression appears to be fully evaluated before any part of any argument expression to its right.

If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated.

15.7.5 Evaluation Order for Other Expressions

The order of evaluation for some expressions is not completely covered by these general rules, because these expressions may raise exceptional conditions at times that must be specified.

See, specifically, the detailed explanations of evaluation order for the following kinds of expressions:

- class instance creation expressions (§15.9.4)
- array creation expressions (§15.10.1)
- method invocation expressions (§15.12.4)
- array access expressions (§15.13.1)
- assignments involving array components (§15.26)

15.8 Primary Expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, class literals, field accesses, method invocations, and array accesses. A parenthesized expression is also treated syntactically as a primary expression.

Primary:

PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:

Literal
Type . class
void . class
this
ClassName . this
(Expression)
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess

15.8.1 Lexical Literals

A literal (§3.10) denotes a fixed, unchanging value.

The following production from §3.10 is repeated here for convenience:

Literal:

IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

The type of a literal is determined as follows:

- The type of an integer literal that ends with `L` or `l` is `long`.

The type of any other integer literal is `int`.

- The type of a floating-point literal that ends with `F` or `f` is `float` and its value must be an element of the float value set (§4.2.3).

The type of any other floating-point literal is `double` and its value must be an element of the double value set.

- The type of a boolean literal is `boolean`.
- The type of a character literal is `char`.

- The type of a string literal is `String`.
- The type of the null literal `null` is the null type; its value is the null reference.

Evaluation of a lexical literal always completes normally.

15.8.2 Class Literals

A *class literal* is an expression consisting of the name of a class, interface, array, or primitive type, or the pseudo-type `void`, followed by a `'.'` and the token `class`.

The type of a class literal, `C.class`, where `C` is the name of a class, interface, or array type, is `Class<C>`.

If `p` is the name of a primitive type, let `B` be the type of an expression of type `p` after boxing conversion (§5.1.7). Then the type of `p.class` is `Class<B>`.

The type of `void.class` is `Class<Void>`.

A class literal evaluates to the `Class` object for the named type (or for `void`) as defined by the defining class loader of the class of the current instance.

It is a compile-time error if any of the following occur:

- The named type is a type variable (§4.4) or a parameterized type (§4.5) or an array whose element type is a type variable or parameterized type.
- The named type does not denote a type that is accessible (§6.6) and in scope (§6.3) at the point where the class literal appears.

15.8.3 `this`

The keyword `this` may be used only in the body of an instance method, instance initializer, or constructor, or in the initializer of an instance variable of a class. If it appears anywhere else, a compile-time error occurs.

When used as a primary expression, the keyword `this` denotes a value that is a reference to the object for which the instance method was invoked (§15.12), or to the object being constructed.

The type of `this` is the class `C` within which the keyword `this` occurs.

At run time, the class of the actual object referred to may be the class `C` or any subclass of `C`.

The keyword `this` is also used in a special explicit constructor invocation statement, which can appear at the beginning of a constructor body (§8.8.7).

15.8.4 Qualified this

Any lexically enclosing instance can be referred to by explicitly qualifying the keyword `this`.

Let C be the class denoted by *ClassName*. Let n be an integer such that C is the n 'th lexically enclosing class (§8.1.3) of the class in which the qualified `this` expression appears.

The value of an expression of the form *ClassName.this* is the n 'th lexically enclosing instance of `this`.

The type of the expression is C .

It is a compile-time error if the current class is not an inner class of class C or C itself.

15.8.5 Parenthesized Expressions

A parenthesized expression is a primary expression whose type is the type of the contained expression and whose value at run time is the value of the contained expression. If the contained expression denotes a variable then the parenthesized expression also denotes that variable.

The use of parentheses affects only the *order* of evaluation, with one fascinating exception.

Consider the case of the smallest possible negative value of type `long`. This value, 9223372036854775808L, is allowed only as an operand of the unary minus operator (§3.10.1). Therefore, enclosing it in parentheses, as in `-(9223372036854775808L)` causes a compile-time error.

In particular, the presence or absence of parentheses around an expression does not (except for the case noted above) affect in any way:

- the choice of value set (§4.2.3) for the value of an expression of type `float` or `double`
- whether a variable is definitely assigned, definitely assigned when `true`, definitely assigned when `false`, definitely unassigned, definitely unassigned when `true`, or definitely unassigned when `false` (Chapter 16, *Definite Assignment*)

15.9 Class Instance Creation Expressions

A class instance creation expression is used to create new objects that are instances of classes.

ClassInstanceCreationExpression:

```
new TypeArgumentsopt ClassOrInterfaceType
                                ( ArgumentListopt ) ClassBodyopt
Primary . new TypeArgumentsopt Identifier TypeArgumentsopt
                                ( ArgumentListopt ) ClassBodyopt
```

ArgumentList:

```
Expression
ArgumentList , Expression
```

A class instance creation expression specifies a class to be instantiated, possibly followed by type arguments (if the class being instantiated is generic (§8.1.2)), followed by (a possibly empty) list of actual value arguments to the constructor.

It is also possible to pass explicit type arguments to the constructor itself (if it is a generic constructor (§8.8.4)). The type arguments to the constructor immediately follow the keyword *new*.

It is a compile-time error if any of the type arguments used in a class instance creation expression are wildcard type arguments (§4.5.1).

Class instance creation expressions have two forms:

- *Unqualified class instance creation expressions* begin with the keyword *new*.
An unqualified class instance creation expression may be used to create an instance of a class, regardless of whether the class is a top-level (§7.6), member (§8.5, §9.5), local (§14.3) or anonymous class (§15.9.5).
- *Qualified class instance creation expressions* begin with a *Primary*.

A qualified class instance creation expression enables the creation of instances of inner member classes and their anonymous subclasses.

Both unqualified and qualified class instance creation expressions may optionally end with a class body. Such a class instance creation expression declares an *anonymous class* (§15.9.5) and creates an instance of it.

We say that a class is *instantiated* when an instance of the class is created by a class instance creation expression. Class instantiation involves determining what class is to be instantiated, what the enclosing instances (if any) of the newly created

instance are, what constructor should be invoked to create the new instance, and what arguments should be passed to that constructor.

15.9.1 Determining the Class being Instantiated

If the class instance creation expression ends in a class body, then the class being instantiated is an anonymous class. Then:

- If the class instance creation expression is an unqualified class instance creation expression, then let *T* be the *ClassOrInterfaceType* after the `new` token.

It is a compile-time error if the class or interface named by *T* is not accessible (§6.6) or if *T* is an enum type (§8.9).

If *T* denotes a class, then an anonymous direct subclass of the class named by *T* is declared. It is a compile-time error if the class denoted by *T* is a `final` class.

If *T* denotes an interface, then an anonymous direct subclass of `Object` that implements the interface named by *T* is declared.

In either case, the body of the subclass is the *ClassBody* given in the class instance creation expression.

The class being instantiated is the anonymous subclass.

- Otherwise, the class instance creation expression is a qualified class instance creation expression. Let *T* be the name of the *Identifier* after the `new` token.

It is a compile-time error if *T* is not the simple name (§6.2) of an accessible (§6.6) non-`final` inner class (§8.1.3) that is a member of the compile-time type of the *Primary*.

It is a compile-time error if *T* is ambiguous (§8.5) or if *T* denotes an enum type.

An anonymous direct subclass of the class named by *T* is declared. The body of the subclass is the *ClassBody* given in the class instance creation expression.

The class being instantiated is the anonymous subclass.

If a class instance creation expression does not declare an anonymous class, then:

- If the class instance creation expression is an unqualified class instance creation expression, then the *ClassOrInterfaceType* must denote a class that is accessible (§6.6) and is not an enum type and not `abstract`, or a compile-time error occurs.

In this case, the class being instantiated is the class denoted by *ClassOrInterfaceType*.

- Otherwise, the class instance creation expression is a qualified class instance creation expression.

It is a compile-time error if *Identifier* is not the simple name (§6.2) of an accessible (§6.6) non-abstract inner class (§8.1.3) *T* that is a member of the compile-time type of the *Primary*.

It is a compile-time error if *Identifier* is ambiguous (§8.5), or if *Identifier* denotes an enum type (§8.9).

The class being instantiated is the class denoted by *Identifier*.

The type of the class instance creation expression is the class type being instantiated.

15.9.2 Determining Enclosing Instances

Let *C* be the class being instantiated, and let *i* be the instance being created. If *C* is an inner class then *i* may have an immediately enclosing instance. The immediately enclosing instance of *i* (§8.1.3) is determined as follows.

If *C* is an anonymous class, then:

- If the class instance creation expression occurs in a static context (§8.1.3), then *i* has no immediately enclosing instance.
- Otherwise, the immediately enclosing instance of *i* is *this*.

If *C* is a local class (§14.3), then let *O* be the innermost lexically enclosing class of *C*. Let *n* be an integer such that *O* is the *n*'th lexically enclosing class of the class in which the class instance creation expression appears. Then:

- If *C* occurs in a static context, then *i* has no immediately enclosing instance.
- Otherwise, if the class instance creation expression occurs in a static context, then a compile-time error occurs.
- Otherwise, the immediately enclosing instance of *i* is the *n*'th lexically enclosing instance of *this* (§8.1.3).

Otherwise, *C* is an inner member class (§8.5), and then:

- If the class instance creation expression is an unqualified class instance creation expression, then:
 - ♦ If the class instance creation expression occurs in a static context, then a compile-time error occurs.

- ◆ Otherwise, if *C* is a member of an enclosing class then let *O* be the innermost lexically enclosing class of which *C* is a member, and let *n* be an integer such that *O* is the *n*'th lexically enclosing class of the class in which the class instance creation expression appears.

The immediately enclosing instance of *i* is the *n*'th lexically enclosing instance of *this*.

- ◆ Otherwise, a compile-time error occurs.
- Otherwise, the class instance creation expression is a qualified class instance creation expression.

The immediately enclosing instance of *i* is the object that is the value of the *Primary* expression.

In addition, if *C* is an anonymous class, and the direct superclass of *C*, *S*, is an inner class, then *i* may have an immediately enclosing instance with respect to *S*. It is determined as follows.

If *S* is a local class (§14.3), then let *O* be the innermost lexically enclosing class of *S*. Let *n* be an integer such that *O* is the *n*'th lexically enclosing class of the class in which the class instance creation expression appears. Then:

- If *S* occurs within a static context, then *i* has no immediately enclosing instance with respect to *S*.
- Otherwise, if the class instance creation expression occurs in a static context, then a compile-time error occurs.
- Otherwise, the immediately enclosing instance of *i* with respect to *S* is the *n*'th lexically enclosing instance of *this*.

Otherwise, *S* is an inner member class (§8.5), and then:

- If the class instance creation expression is an unqualified class instance creation expression, then:
 - ◆ If the class instance creation expression occurs in a static context, then a compile-time error occurs.
 - ◆ Otherwise, if *S* is a member of an enclosing class then let *O* be the innermost lexically enclosing class of which *S* is a member, and let *n* be an integer such that *O* is the *n*'th lexically enclosing class of the class in which the class instance creation expression appears.

The immediately enclosing instance of *i* with respect to *S* is the *n*'th lexically enclosing instance of *this*.

- ♦ Otherwise, a compile-time error occurs.
- Otherwise, the class instance creation expression is a qualified class instance creation expression.

The immediately enclosing instance of *i* with respect to *S* is the object that is the value of the *Primary* expression.

15.9.3 Choosing the Constructor and its Arguments

Let *C* be the class type being instantiated. To create an instance of *C*, *i*, a constructor of *C* is chosen at compile-time by the following rules.

First, the actual arguments to the constructor invocation are determined:

- If *C* is an anonymous class, and the direct superclass of *C*, *S*, is an inner class, then:
 - ♦ If *S* is a local class and *S* occurs in a static context, then the arguments in the argument list, if any, are the arguments to the constructor, in the order they appear in the expression.
 - ♦ Otherwise, the immediately enclosing instance of *i* with respect to *S* is the first argument to the constructor, followed by the arguments in the argument list of the class instance creation expression, if any, in the order they appear in the expression.
- Otherwise the arguments in the argument list, if any, are the arguments to the constructor, in the order they appear in the expression.

Once the actual arguments have been determined, they are used to select a constructor of *C*, using the same rules as for method invocations (§15.12).

As in method invocations, a compile-time method matching error occurs if there is no unique most-specific constructor that is both applicable and accessible.

Note that the type of the class instance creation expression may be an anonymous class type, in which case the constructor being invoked is an anonymous constructor (§15.9.5.1).

15.9.4 Run-time Evaluation of Class Instance Creation Expressions

At run time, evaluation of a class instance creation expression is as follows.

First, if the class instance creation expression is a qualified class instance creation expression, the qualifying primary expression is evaluated. If the qualifying expression evaluates to `null`, a `NullPointerException` is raised, and the class

instance creation expression completes abruptly. If the qualifying expression completes abruptly, the class instance creation expression completes abruptly for the same reason.

Next, space is allocated for the new class instance. If there is insufficient space to allocate the object, evaluation of the class instance creation expression completes abruptly by throwing an `OutOfMemoryError`.

The new object contains new instances of all the fields declared in the specified class type and all its superclasses. As each new field instance is created, it is initialized to its default value (§4.12.5).

Next, the actual arguments to the constructor are evaluated, left-to-right. If any of the argument evaluations completes abruptly, any argument expressions to its right are not evaluated, and the class instance creation expression completes abruptly for the same reason.

Next, the selected constructor of the specified class type is invoked. This results in invoking at least one constructor for each superclass of the class type. This process can be directed by explicit constructor invocation statements (§8.8) and is described in detail in §12.5.

The value of a class instance creation expression is a reference to the newly created object of the specified class. Every time the expression is evaluated, a fresh object is created.

15.9.5 Anonymous Class Declarations

An anonymous class declaration is automatically derived from a class instance creation expression by the Java compiler.

An anonymous class is never `abstract` (§8.1.1.1).

An anonymous class is always an inner class (§8.1.3); it is never `static` (§8.1.1, §8.5.2).

An anonymous class is always implicitly `final` (§8.1.1.2).

15.9.5.1 Anonymous Constructors

An anonymous class cannot have an explicitly declared constructor. Instead, the Java compiler must automatically provide an anonymous constructor for the anonymous class. The form of the anonymous constructor of an anonymous class *C* with direct superclass *S* is as follows:

- If S is not an inner class, or if S is a local class that occurs in a static context, then the anonymous constructor has one formal parameter for each actual argument to the class instance creation expression in which C is declared.

The actual arguments to the class instance creation expression are used to determine a constructor cs of S , using the same rules as for method invocations (§15.12).

The type of each formal parameter of the anonymous constructor must be identical to the corresponding formal parameter of cs .

The body of the constructor consists of an explicit constructor invocation (§8.8.7.1) of the form `super(...)`, where the actual arguments are the formal parameters of the constructor, in the order they were declared.

- Otherwise, the first formal parameter of the constructor of C represents the value of the immediately enclosing instance of i with respect to S . The type of this parameter is the class type that immediately encloses the declaration of S .

The constructor has an additional formal parameter for each actual argument to the class instance creation expression that declared the anonymous class. The n 'th formal parameter e corresponds to the $n-1$ 'th actual argument.

The actual arguments to the class instance creation expression are used to determine a constructor cs of S , using the same rules as for method invocations (§15.12).

The type of each formal parameter of the anonymous constructor must be identical to the corresponding formal parameter of cs .

The body of the constructor consists of an explicit constructor invocation (§8.8.7.1) of the form `o.super(...)`, where o is the first formal parameter of the constructor, and the actual arguments are the subsequent formal parameters of the constructor, in the order they were declared.

In all cases, the `throws` clause of an anonymous constructor must list all the checked exceptions thrown by the explicit superclass constructor invocation statement contained within the anonymous constructor, and all checked exceptions thrown by any instance initializers or instance variable initializers of the anonymous class.

Note that it is possible for the signature of the anonymous constructor to refer to an inaccessible type (for example, if such a type occurred in the signature of the superclass constructor cs). This does not, in itself, cause any errors at either compile time or run time.

15.10 Array Creation Expressions

An array creation expression is used to create new arrays (Chapter 10, *Arrays*).

ArrayCreationExpression:

```
new PrimitiveType DimExprs Dimsopt
new ClassOrInterfaceType DimExprs Dimsopt
new PrimitiveType Dims ArrayInitializer
new ClassOrInterfaceType Dims ArrayInitializer
```

DimExprs:

```
DimExpr
DimExprs DimExpr
```

DimExpr:

```
[ Expression ]
```

Dims:

```
[ ]
Dims [ ]
```

An array creation expression creates an object that is a new array whose elements are of the type specified by the *PrimitiveType* or *ClassOrInterfaceType*.

It is a compile-time error if the *ClassOrInterfaceType* does not denote a type that is reifiable (§4.7). Otherwise, the *ClassOrInterfaceType* may name any named reference type, even an abstract class type (§8.1.1.1) or an interface type (Chapter 9, *Interfaces*).

The type of the creation expression is an array type that can denoted by a copy of the creation expression from which the new keyword and every *DimExpr* expression and array initializer have been deleted.

The type of each dimension expression within a *DimExpr* must be a type that is convertible (§5.1.8) to an integral type, or a compile-time error occurs.

Each expression undergoes unary numeric promotion (§5.6.1). The promoted type must be `int`, or a compile-time error occurs.

If an array initializer is provided, the newly allocated array will be initialized with the values provided by the array initializer as described in §10.6.

15.10.1 Run-time Evaluation of Array Creation Expressions

At run time, evaluation of an array creation expression behaves as follows.

If there are no dimension expressions, then there must be an array initializer. The value of the array initializer is the value of the array creation expression. Otherwise:

First, the dimension expressions are evaluated, left-to-right. If any of the expression evaluations completes abruptly, the expressions to the right of it are not evaluated.

Next, the values of the dimension expressions are checked. If the value of any *DimExpr* expression is less than zero, then an `NegativeArraySizeException` is thrown.

Next, space is allocated for the new array. If there is insufficient space to allocate the array, evaluation of the array creation expression completes abruptly by throwing an `OutOfMemoryError`.

Then, if a single *DimExpr* appears, a one-dimensional array is created of the specified length, and each component of the array is initialized to its default value (§4.12.5).

If an array creation expression contains n *DimExpr* expressions, then it effectively executes a set of nested loops of depth $n-1$ to create the implied arrays of arrays.

A multidimensional array need not have arrays of the same length at each level.

In an array creation expression (§15.10), there may be one or more dimension expressions, each within brackets. Each dimension expression is fully evaluated before any part of any dimension expression to its right.

If evaluation of a dimension expression completes abruptly, no part of any dimension expression to its right will appear to have been evaluated.

If evaluation of an array creation expression finds there is insufficient memory to perform the creation operation, then an `OutOfMemoryError` is thrown. If the array creation expression does not have an array initializer, then this check occurs only after evaluation of all dimension expressions has completed normally. If the array creation expression does have an array initializer, then an `OutOfMemoryError` can occur when an object of reference type is allocated during evaluation of a variable initializer expression, or when space is allocated for an array to hold the values of a (possibly nested) array initializer.

15.11 Field Access Expressions

A field access expression may access a field of an object or array, a reference to which is the value of either an expression or the special keyword `super`.

FieldAccess:

Primary . *Identifier*

`super` . *Identifier*

ClassName . `super` . *Identifier*

The meaning of a field access expression is determined using the same rules as for qualified names (§6.5.6.2), but limited by the fact that an expression cannot denote a package, class type, or interface type.

15.11.1 Field Access Using a Primary

The type of the *Primary* must be a reference type *T*, or a compile-time error occurs.

The meaning of the field access expression is determined as follows:

- If the identifier names several accessible (§6.6) member fields of type *T*, then the field access is ambiguous and a compile-time error occurs.
- If the identifier does not name an accessible member field of type *T*, then the field access is undefined and a compile-time error occurs.
- Otherwise, the identifier names a single accessible member field of type *T* and the type of the field access expression is the type of the member field after capture conversion (§5.1.10).

At run-time, the result of the field access expression is computed as follows: (assuming that the program is correct with respect to definite assignment analysis, i.e. every blank `final` variable is definitely assigned before access)

- ♦ If the field is `static`:
 - ❖ The *Primary* expression is evaluated, and the result is discarded. If evaluation of the *Primary* expression completes abruptly, the field access expression completes abruptly for the same reason.
 - ❖ If the field is a non-blank `final`, then the result is the value of the specified class variable in the class or interface that is the type of the *Primary* expression.

- ❖ If the field is not `final`, or is a blank `final` and the field access occurs in a constructor, then the result is a variable, namely, the specified class variable in the class that is the type of the *Primary* expression.
- ♦ If the field is not `static`:
 - ❖ The *Primary* expression is evaluated. If evaluation of the *Primary* expression completes abruptly, the field access expression completes abruptly for the same reason.
 - ❖ If the value of the *Primary* is `null`, then a `NullPointerException` is thrown.
 - ❖ If the field is a non-blank `final`, then the result is the value of the specified instance variable in the object referenced by the value of the *Primary*.
 - ❖ If the field is not `final`, or is a blank `final` and the field access occurs in a constructor, then the result is a variable, namely, the specified instance variable in the object referenced by the value of the *Primary*.

Note, specifically, that only the type of the *Primary* expression, not the class of the actual object referred to at run time, is used in determining which field to use.

15.11.2 Accessing Superclass Members using `super`

The form `super.Identifier` refers to the field named *Identifier* of the current object, but with the current object viewed as an instance of the superclass of the current class. The form `T.super.Identifier` refers to the field named *Identifier* of the lexically enclosing instance corresponding to *T*, but with that instance viewed as an instance of the superclass of *T*.

The forms using the keyword `super` are valid only in an instance method, instance initializer or constructor, or in the initializer of an instance variable of a class. These are exactly the same situations in which the keyword `this` may be used (§15.8.3).

It is a compile-time error if the forms using the keyword `super` appear in the declaration of class `Object`, since `Object` has no superclass.

If a field access expression `super.name` appears within class *C*, and the immediate superclass of *C* is class *S*, then `super.name` is treated exactly as if it had been the expression `this.name` in the body of class *S*. Thus it can access the field *name* that is visible in class *S*, even if that field is hidden by a declaration of a field *name* in class *C*.

If a field access expression *T.super.name* appears within class *C*, and the immediate superclass of the class denoted by *T* is a class whose fully qualified name is *S*, then *T.super.name* is treated exactly as if it had been the expression *this.name* in the body of class *S*. Thus it can access the field *name* that is visible in class *S*, even if that field is hidden by a declaration of a field *name* in class *T*.

It is a compile-time error if the current class is not an inner class of class *T* or *T* itself.

15.12 Method Invocation Expressions

A method invocation expression is used to invoke a class or instance method.

MethodInvocation:

MethodName (*ArgumentList*_{opt})

Primary . *NonWildTypeArguments*_{opt} *Identifier* (*ArgumentList*_{opt})

super . *NonWildTypeArguments*_{opt} *Identifier* (*ArgumentList*_{opt})

ClassName . *super* . *NonWildTypeArguments*_{opt} *Identifier* (*ArgumentList*_{opt})

TypeName . *NonWildTypeArguments* *Identifier* (*ArgumentList*_{opt})

The definition of *ArgumentList* from §15.9 is repeated here for convenience:

ArgumentList:

Expression

ArgumentList , *Expression*

Resolving a method name at compile time is more complicated than resolving a field name because of the possibility of method overloading. Invoking a method at run time is also more complicated than accessing a field because of the possibility of instance method overriding.

Determining the method that will be invoked by a method invocation expression involves several steps. The following three sections describe the compile-time processing of a method invocation; the determination of the type of the method invocation expression is described in §15.12.3.

15.12.1 Compile-Time Step 1: Determine Class or Interface to Search

The first step in processing a method invocation at compile time is to figure out the name of the method to be invoked and which class or interface to check for definitions of methods of that name. There are several cases to consider, depending on the form that precedes the left parenthesis, as follows.

- If the form is *MethodName*, then there are three subcases:
 - ♦ If it is a simple name, that is, just an *Identifier*, then the name of the method is the *Identifier*.

If the *Identifier* appears within the scope (§6.3) of a visible method declaration with that name, then:

- ♦ If there is an enclosing type declaration of which that method is a member, let T be the innermost such type declaration. The class or interface to search is T .
- ♦ Otherwise, the visible method declaration may be in scope due to one or more single-static-import (§7.5.3) or static-import-on-demand (§7.5.4) declarations. There is no class or interface to search, as the method to be invoked is determined later (§15.12.2).
- ♦ If it is a qualified name of the form *TypeName* . *Identifier*, then the name of the method is the *Identifier* and the class to search is the one named by the *TypeName*.

If *TypeName* is the name of an interface rather than a class, then a compile-time error occurs, because this form can invoke only `static` methods and interfaces have no `static` methods.

- ♦ In all other cases, the qualified name has the form *FieldName* . *Identifier*.

The name of the method is the *Identifier* and the class or interface to search is the declared type T of the field named by the *FieldName*, if T is a class or interface type, or the upper bound of T if T is a type variable.

- If the form is *Primary* . *NonWildTypeArguments*_{opt} *Identifier*, then the name of the method is the *Identifier*.

Let T be the type of the *Primary* expression. The class or interface to be searched is T if T is a class or interface type, or the upper bound of T if T is a type variable.

It is a compile-time error if T is not a reference type.

- If the form is `super` . *NonWildTypeArguments*_{opt} *Identifier*, then the name of the method is the *Identifier* and the class to be searched is the superclass of the class whose declaration contains the method invocation.

Let T be the type declaration immediately enclosing the method invocation. It is a compile-time error if T is the class `Object` or T is an interface.

- If the form is *ClassName* . *super* . *NonWildTypeArguments*_{opt} *Identifier*, then the name of the method is the *Identifier* and the class to be searched is the superclass of the class *C* denoted by *ClassName*.

It is a compile-time error if *C* is not a lexically enclosing class of the current class.

It is a compile-time error if *C* is the class `Object`.

Let *T* be the type declaration immediately enclosing the method invocation. It is a compile-time error if *T* is the class `Object` or *T* is an interface.

- If the form is *TypeName* . *NonWildTypeArguments* *Identifier*, then the name of the method is the *Identifier* and the class to be searched is the class *C* denoted by *TypeName*.

If *TypeName* is the name of an interface rather than a class, then a compile-time error occurs, because this form can invoke only `static` methods and interfaces have no `static` methods.

15.12.2 Compile-Time Step 2: Determine Method Signature

The second step searches the type determined in the previous step for member methods. This step uses the name of the method and the types of the argument expressions to locate methods that are both *accessible* and *applicable*, that is, declarations that can be correctly invoked on the given arguments.

There may be more than one such method, in which case the *most specific* one is chosen. The descriptor (signature plus return type) of the most specific method is one used at run time to perform the method dispatch.

A method is *applicable* if it is either applicable by subtyping (§15.12.2.2), applicable by method invocation conversion (§15.12.2.3), or it is an applicable variable arity method (§15.12.2.4).

The process of determining applicability begins by determining the potentially applicable methods (§15.12.2.1).

The remainder of the process is split into three phases, to ensure compatibility with versions of the Java programming language prior to Java SE 5.0. The phases are:

1. The first phase (§15.12.2.2) performs overload resolution without permitting boxing or unboxing conversion, or the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the second phase.

2. The second phase (§15.12.2.3) performs overload resolution while allowing boxing and unboxing, but still precludes the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the third phase.
3. The third phase (§15.12.2.4) allows overloading to be combined with variable arity methods, boxing, and unboxing.

Deciding whether a method is applicable will, in the case of generic methods (§8.4.4), require that type arguments be determined. Type arguments may be passed explicitly or implicitly. If they are passed implicitly, they must be inferred (§15.12.2.7) from the types of the argument expressions.

If several applicable methods have been identified during one of the three phases of applicability testing, then the most specific one is chosen, as specified in section §15.12.2.5.

15.12.2.1 Identify Potentially Applicable Methods

The class or interface determined by compile-time step 1 (§15.12.1) is searched for all member methods that are potentially applicable to this method invocation; members inherited from superclasses and superinterfaces are included in this search.

In addition, if the method invocation has, before the left parenthesis, a *MethodName* of the form *Identifier*, then the search process also examines all member methods that are (a) imported by single-static-import declarations (§7.5.3) and static-import-on-demand declarations (§7.5.4) within the compilation unit (§7.3) within which the method invocation occurs, and (b) not shadowed (§6.4.1) at the place where the method invocation appears, to determine if they are potentially applicable.

A member method is *potentially applicable* to a method invocation if and only if all of the following are true:

- The name of the member is identical to the name of the method in the method invocation.
- The member is accessible (§6.6) to the class or interface in which the method invocation appears.
- If the member is a variable arity method with arity n , the arity of the method invocation is greater or equal to $n-1$.
- If the member is a fixed arity method with arity n , the arity of the method invocation is equal to n .

- If the method invocation includes explicit type arguments, and the member is a generic method, then the number of type arguments is equal to the number of type parameters of the method.

If the search does not yield at least one method that is potentially applicable, then a compile-time error occurs.

15.12.2.2 Phase 1: Identify Matching Arity Methods Applicable by Subtyping

Let m be a potentially applicable method (§15.12.2.1), let $e_1, \dots, e_n$ be the actual argument expressions of the method invocation, and let A_i be the type of e_i ($1 \leq i \leq n$). Then:

- If m is a generic method, then let $F_1 \dots F_n$ be the types of the formal parameters of m , and let $R_1 \dots R_p$ ($p \geq 1$) be the type parameters of m , and let B_l be the declared bound of R_l ($1 \leq l \leq p$). Then:
 - ♦ If the method invocation does not provide explicit type arguments, then let $U_1 \dots U_p$ be the type arguments inferred (§15.12.2.7) for this invocation of m , using a set of initial constraints consisting of the constraints $A_i << F_i$ ($1 \leq i \leq n$) for each actual argument expression e_i whose type is a reference type.
 - ♦ Otherwise, let $U_1 \dots U_p$ be the explicit type arguments given in the method invocation.

Then let $S_i = F_i[R_1=U_1, \dots, R_p=U_p]$ ($1 \leq i \leq n$) be the types inferred for the formal parameters of m .

- Otherwise, let $S_1 \dots S_n$ be the types of the formal parameters of m .

The method m is *applicable by subtyping* if and only if both of the following conditions hold:

- For $1 \leq i \leq n$, either:
 - ♦ $A_i <: S_i$ (§4.10), or
 - ♦ A_i is convertible to some type C_i by unchecked conversion (§5.1.9), and $C_i <: S_i$.
- If m is a generic method as described above, then $U_l <: B_l[R_1=U_1, \dots, R_p=U_p]$ ($1 \leq l \leq p$).

If no method applicable by subtyping is found, the search for applicable methods continues with phase 2 (§15.12.2.3). Otherwise, the most specific method (§15.12.2.5) is chosen among the methods that are applicable by subtyping.

15.12.2.3 Phase 2: Identify Matching Arity Methods Applicable by Method Invocation Conversion

Let m be a potentially applicable method (§15.12.2.1), let $e_1, \dots, e_n$ be the actual argument expressions of the method invocation, and let A_i be the type of e_i ($1 \leq i \leq n$). Then:

- If m is a generic method, then let $F_1 \dots F_n$ be the types of the formal parameters of m , and let $R_1 \dots R_p$ ($p \geq 1$) be the type parameters of m , and let B_l be the declared bound of R_l ($1 \leq l \leq p$). Then:
 - ♦ If the method invocation does not provide explicit type arguments, then let $U_1 \dots U_p$ be the type arguments inferred (§15.12.2.7) for this invocation of m , using a set of initial constraints consisting of the constraints $A_i << F_i$ ($1 \leq i \leq n$).
 - ♦ Otherwise, let $U_1 \dots U_p$ be the explicit type arguments given in the method invocation.

Then let $S_i = F_i[R_1=U_1, \dots, R_p=U_p]$ ($1 \leq i \leq n$) be the types inferred for the formal parameters of m .

- Otherwise, let $S_1 \dots S_n$ be the types of the formal parameters of m .

The method m is *applicable by method invocation conversion* if and only if both of the following conditions hold:

- For $1 \leq i \leq n$, the type of e_i , A_i , can be converted by method invocation conversion (§5.3) to S_i .
- If m is a generic method as described above, then $U_l <: B_l[R_1=U_1, \dots, R_p=U_p]$ ($1 \leq l \leq p$).

If no method applicable by method invocation conversion is found, the search for applicable methods continues with phase 3 (§15.12.2.4). Otherwise, the most specific method (§15.12.2.5) is chosen among the methods that are applicable by method invocation conversion.

15.12.2.4 Phase 3: Identify Applicable Variable Arity Methods

Let m be a potentially applicable method (§15.12.2.1) with variable arity, let $e_1, \dots, e_k$ be the actual argument expressions of the method invocation, and let A_i be the type of e_i ($1 \leq i \leq k$). Then:

- If m is a generic method, then let $F_1 \dots F_n$ ($1 \leq n \leq k+1$) be the types of the formal parameters of m , where $F_n = T[]$ for some type T , and let $R_1 \dots R_p$ ($p \geq 1$) be the type parameters of m , and let B_l be the declared bound of R_l ($1 \leq l \leq p$). Then:

- ♦ If the method invocation does not provide explicit type arguments then let $U_1 \dots U_p$ be the type arguments inferred (§15.12.2.7) for this invocation of m , using a set of initial constraints consisting of the constraints $A_i << F_i$ ($1 \leq i < n$) and the constraints $A_j << T$ ($n \leq j \leq k$).
- ♦ Otherwise let $U_1 \dots U_p$ be the explicit type arguments given in the method invocation.

Then let $S_i = F_i[R_1=U_1, \dots, R_p=U_p]$ ($1 \leq i \leq n$) be the types inferred for the formal parameters of m .

- Otherwise, let $S_1 \dots S_n$ (where $n \leq k+1$) be the types of the formal parameters of m .

The method m is an *applicable variable-arity method* if and only if all of the following conditions hold:

- For $1 \leq i < n$, the type of e_i , A_i , can be converted by method invocation conversion to S_i .
- If $k \geq n$, then for $n \leq i \leq k$, the type of e_i , A_i , can be converted by method invocation conversion to the component type of S_n .
- If $k \neq n$, or if $k = n$ and A_n cannot be converted by method invocation conversion to $S_n[]$, then the type which is the erasure of S_n is accessible at the point of invocation.
- If m is a generic method as described above, then $U_l <: B_l[R_1=U_1, \dots, R_p=U_p]$ ($1 \leq l \leq p$).

If no applicable variable arity method is found, a compile-time error occurs.

Otherwise, the most specific method (§15.12.2.5) is chosen among the applicable variable-arity methods.

15.12.2.5 Choosing the Most Specific Method

If more than one member method is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the run-time method dispatch. The Java programming language uses the rule that the *most specific* method is chosen.

The informal intuition is that one method is more specific than another if any invocation handled by the first method could be passed on to the other one without a compile-time type error.

One fixed-arity member method named m is *more specific* than another member method of the same name and arity if all of the following conditions hold:

- The declared types of the parameters of the first member method are $T_1, \dots, T_n$.
- The declared types of the parameters of the other method are $U_1, \dots, U_n$.
- If the second method is generic, then let $R_1 \dots R_p$ ($p \geq 1$) be its type parameters, let B_l be the declared bound of R_l ($1 \leq l \leq p$), let $A_1 \dots A_p$ be the type arguments inferred (§15.12.2.7) for this invocation under the initial constraints $T_i << U_i$ ($1 \leq i \leq n$), and let $S_i = U_i[R_1=A_1, \dots, R_p=A_p]$ ($1 \leq i \leq n$).

Otherwise, let $S_i = U_i$ ($1 \leq i \leq n$).

- For all j from 1 to n , $T_j <: S_j$.
- If the second method is a generic method as described above, then $A_l <: B_l[R_1=A_1, \dots, R_p=A_p]$ ($1 \leq l \leq p$).

In addition, one variable arity member method named m is more specific than another variable arity member method of the same name if either:

1. One member method has n parameters and the other has k parameters, where $n \geq k$, and:

- The types of the parameters of the first member method are $T_1, \dots, T_{n-1}, T_n[]$.
- The types of the parameters of the other method are $U_1, \dots, U_{k-1}, U_k[]$.
- If the second method is generic then let $R_1 \dots R_p$ ($p \geq 1$) be its type parameters, let B_l be the declared bound of R_l ($1 \leq l \leq p$), let $A_1 \dots A_p$ be the type arguments inferred (§15.12.2.7) for this invocation under the initial constraints $T_i << U_i$ ($1 \leq i \leq k-1$) and $T_i << U_k$ ($k \leq i \leq n$), and let $S_i = U_i[R_1=A_1, \dots, R_p=A_p]$ ($1 \leq i \leq k$).

Otherwise, let $S_i = U_i$ ($1 \leq i \leq k$).

- For all j from 1 to $k-1$, $T_j <: S_j$, and,
- For all j from k to n , $T_j <: S_k$, and,
- If the second method is a generic method as described above, then $A_l <: B_l[R_1=A_1, \dots, R_p=A_p]$ ($1 \leq l \leq p$).

2. One member method has k parameters and the other has n parameters, where $n \geq k$, and:

- The types of the parameters of the first method are $U_1, \dots, U_{k-1}, U_k[]$.
- The types of the parameters of the other method are $T_1, \dots, T_{n-1}, T_n[]$.
- If the second method is generic, then let $R_1 \dots R_p$ ($p \geq 1$) be its type parameters, let B_l be the declared bound of R_l ($1 \leq l \leq p$), let $A_1 \dots A_p$ be the type arguments

inferred (§15.12.2.7) for this invocation under the initial constraints $U_i << T_i$ ($1 \leq i \leq k-1$) and $U_k << T_i$ ($k \leq i \leq n$), and let $S_i = T_i[R_1=A_1, \dots, R_p=A_p]$ ($1 \leq i \leq n$).

Otherwise, let $S_i = T_i$ ($1 \leq i \leq n$).

- For all j from 1 to $k-1$, $U_j <: S_j$, and,
- For all j from k to n , $U_k <: S_j$, and,
- If the second method is a generic method as described above, then $A_l <: B_l[R_1=A_1, \dots, R_p=A_p]$ ($1 \leq l \leq p$).

The above conditions are the only circumstances under which one method may be more specific than another.

A method m_1 is *strictly more specific* than another method m_2 if and only if m_1 is more specific than m_2 and m_2 is not more specific than m_1 .

A method is said to be *maximally specific* for a method invocation if it is accessible and applicable and there is no other method that is applicable and accessible that is strictly more specific.

If there is exactly one maximally specific method, then that method is in fact the *most specific method*; it is necessarily more specific than any other accessible method that is applicable. It is then subjected to some further compile-time checks as described in §15.12.3.

It is possible that no method is the most specific, because there are two or more methods that are maximally specific. In this case:

- If all the maximally specific methods have override-equivalent (§8.4.2) signatures, then:
 - ♦ If exactly one of the maximally specific methods is not declared abstract, it is the most specific method.
 - ♦ Otherwise, if all the maximally specific methods are declared abstract, and the signatures of all of the maximally specific methods have the same erasure (§4.6), then the most specific method is chosen arbitrarily among the subset of the maximally specific methods that have the most specific return type.

However, the most specific method is considered to throw a checked exception if and only if that exception or its erasure is declared in the throws clauses of each of the maximally specific methods.

- Otherwise, we say that the method invocation is *ambiguous*, and a compile-time error occurs.

15.12.2.6 Method Result and Throws Types

The result type of the chosen method is determined as follows:

- If the chosen method is declared with a return type of `void`, then the result is `void`.
- Otherwise, if unchecked conversion was necessary for the method to be applicable, then the result type is the erasure (§4.6) of the method's declared return type.
- Otherwise, if the chosen method is generic, then for $1 \leq i \leq n$, let F_i be the formal type parameters of the method, let A_i be the actual type arguments inferred for the method invocation, and let R be the return type of the chosen method.

The result type is obtained by applying capture conversion (§5.1.10) to $R[F_1=A_1, \dots, F_n=A_n]$.

- Otherwise, the result type is obtained by applying capture conversion (§5.1.10) to the return type of the chosen method.

The exception types of the `throws` clause of the chosen method are determined as follows:

- If unchecked conversion was necessary for the method to be applicable, then the `throws` clause is composed of the erasure (§4.6) of the types in the method's declared `throws` clause.
- Otherwise, if the method being invoked is generic, then for $1 \leq i \leq n$, let F_i be the type parameters of the method, let A_i be the type arguments inferred for the method invocation, and let E_j ($1 \leq j \leq m$) be the exception types declared in the `throws` clause of the method being invoked.

The `throws` clause consists of the types $E_j[F_1=A_1, \dots, F_n=A_n]$.

- Otherwise, the type of the `throws` clause is the type given in the method declaration.

15.12.2.7 Inferring Type Arguments Based on Actual Arguments

In this section, we describe the process of inferring type arguments for method and constructor invocations. This process is invoked as a subroutine when testing for method (or constructor) applicability (§15.12.2.2 - §15.12.2.4).

We use the following notational conventions in this section:

- Type expressions are represented using the letters A , F , U , V , and W . The letter A is only used to denote the type of an actual argument, and F is only used to denote the type of a formal parameter.
- Type parameters are represented using the letters S and T .
- Arguments to parameterized types are represented using the letters X and Y .
- Generic type declarations are represented using the letters G and H .

Inference begins with a set of initial constraints of the form $A \ll F$, $A = F$, or $A \gg F$, where $U \ll V$ indicates that type U is convertible to type V by method invocation conversion (§5.3), and $U \gg V$ indicates that type V is convertible to type U by method invocation conversion.

These constraints are then reduced to a set of simpler constraints of the forms $T \rightarrow X$, $T = X$, or $T \leftarrow X$, where T is a type parameter of the method. This reduction is achieved by the procedure given below.

Given a constraint of the form $A \ll F$, $A = F$, or $A \gg F$:

If F does not involve a type parameter T_j then no constraint is implied on T_j .

Otherwise, F involves a type parameter T_j .

If A is the type of `null`, no constraint is implied on T_j .

Otherwise, if the constraint has the form $A \ll F$:

- If A is a primitive type, then A is converted to a reference type U via boxing conversion and this algorithm is applied recursively to the constraint $U \ll F$.
- Otherwise, if $F = T_j$, then the constraint $T_j \rightarrow A$ is implied.
- If $F = U[]$, where the type U involves T_j , then if A is an array type $V[]$, or a type variable with an upper bound that is an array type $V[]$, where V is a reference type, this algorithm is applied recursively to the constraint $V \ll U$.
- If F has the form $G\langle \dots, Y_{k-1}, U, Y_{k+1}, \dots \rangle$, where U is a type expression that involves T_j , then if A has a supertype of the form $G\langle \dots, X_{k-1}, V, X_{k+1}, \dots \rangle$ where V is a type expression, this algorithm is applied recursively to the constraint $V = U$.
- If F has the form $G\langle \dots, Y_{k-1}, ? \text{ extends } U, Y_{k+1}, \dots \rangle$, where U involves T_j , then if A has a supertype that is one of:
 - ♦ $G\langle \dots, X_{k-1}, V, X_{k+1}, \dots \rangle$, where V is a type expression. Then this algorithm is applied recursively to the constraint $V \ll U$.
 - ♦ $G\langle \dots, X_{k-1}, ? \text{ extends } V, X_{k+1}, \dots \rangle$. Then this algorithm is applied recursively to the constraint $V \ll U$.

- ♦ Otherwise, no constraint is implied on T_j .
- If F has the form $G<..., Y_{k-1}, ? \text{ super } U, Y_{k+1}, ...>$, where U involves T_j , then if A has a supertype that is one of:
 - ♦ $G<..., X_{k-1}, V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V >> U$.
 - ♦ $G<..., X_{k-1}, ? \text{ super } V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V >> U$.
 - ♦ Otherwise, no constraint is implied on T_j .
- Otherwise, no constraint is implied on T_j .

Otherwise, if the constraint has the form $A = F$:

- If $F = T_j$, then the constraint $T_j = A$ is implied.
- If $F = U[]$ where the type U involves T_j , then if A is an array type $V[]$, or a type variable with an upper bound that is an array type $V[]$, where V is a reference type, this algorithm is applied recursively to the constraint $V = U$.
- If F has the form $G<..., Y_{k-1}, U, Y_{k+1}, ...>$, where U is type expression that involves T_j , then if A is of the form $G<..., X_{k-1}, V, X_{k+1}, ...>$ where V is a type expression, this algorithm is applied recursively to the constraint $V = U$.
- If F has the form $G<..., Y_{k-1}, ? \text{ extends } U, Y_{k+1}, ...>$, where U involves T_j , then if A is one of:
 - ♦ $G<..., X_{k-1}, ? \text{ extends } V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V = U$.
 - ♦ Otherwise, no constraint is implied on T_j .
- If F has the form $G<..., Y_{k-1}, ? \text{ super } U, Y_{k+1}, ...>$, where U involves T_j , then if A is one of:
 - ♦ $G<..., X_{k-1}, ? \text{ super } V, X_{k+1}, ...>$. Then this algorithm is applied recursively to the constraint $V = U$.
 - ♦ Otherwise, no constraint is implied on T_j .
- Otherwise, no constraint is implied on T_j .

Otherwise, if the constraint has the form $A >> F$:

- If $F = T_j$, then the constraint $T_j <: A$ is implied.

- If $F = U[]$, where the type U involves T_j , then if A is an array type $V[]$, or a type variable with an upper bound that is an array type $V[]$, where V is a reference type, this algorithm is applied recursively to the constraint $V >> U$. Otherwise, no constraint is implied on T_j .
- If F has the form $G<..., Y_{k-1}, U, Y_{k+1}, ...>$, where U is a type expression that involves T_j , then:
 - ♦ If A is an instance of a non-generic type, then no constraint is implied on T_j .
 - ♦ If A is an invocation of a generic type declaration H , where H is either G or superclass or superinterface of G , then:
 - ✧ If $H \neq G$, then let $S_1, ..., S_n$ be the type parameters of G , and let $H<U_1, ..., U_l>$ be the unique invocation of H that is a supertype of $G<S_1, ..., S_n>$, and let $V = H<U_1, ..., U_l>[S_k=U]$. Then, if $V := F$ this algorithm is applied recursively to the constraint $A >> V$.
 - ✧ Otherwise, if A is of the form $G<..., X_{k-1}, W, X_{k+1}, ...>$, where W is a type expression, this algorithm is applied recursively to the constraint $W = U$.
 - ✧ Otherwise, if A is of the form $G<..., X_{k-1}, ? \text{ extends } W, X_{k+1}, ...>$, this algorithm is applied recursively to the constraint $W >> U$.
 - ✧ Otherwise, if A is of the form $G<..., X_{k-1}, ? \text{ super } W, X_{k+1}, ...>$, this algorithm is applied recursively to the constraint $W << U$.
 - ✧ Otherwise, no constraint is implied on T_j .
- If F has the form $G<..., Y_{k-1}, ? \text{ extends } U, Y_{k+1}, ...>$, where U is a type expression that involves T_j , then:
 - ♦ If A is an instance of a non-generic type, then no constraint is implied on T_j .
 - ♦ If A is an invocation of a generic type declaration H , where H is either G or superclass or superinterface of G , then:
 - ✧ If $H \neq G$, then let $S_1, ..., S_n$ be the type parameters of G , and let $H<U_1, ..., U_l>$ be the unique invocation of H that is a supertype of $G<S_1, ..., S_n>$, and let $V = H<? \text{ extends } U_1, ..., ? \text{ extends } U_l>[S_k=U]$. Then this algorithm is applied recursively to the constraint $A >> V$.
 - ✧ Otherwise, if A is of the form $G<..., X_{k-1}, ? \text{ extends } W, X_{k+1}, ...>$, this algorithm is applied recursively to the constraint $W >> U$.
 - ✧ Otherwise, no constraint is implied on T_j .
- If F has the form $G<..., Y_{k-1}, ? \text{ super } U, Y_{k+1}, ...>$, where U is a type expression that involves T_j , then A is either:

- ♦ If A is an instance of a non-generic type, then no constraint is implied on T_j .
- ♦ If A is an invocation of a generic type declaration H , where H is either G or superclass or superinterface of G , then:
 - ♦ If $H \neq G$, then let $S_1, \dots, S_n$ be the type parameters of G , and let $H < U_1, \dots, U_l >$ be the unique invocation of H that is a supertype of $G < S_1, \dots, S_n >$, and let $V = H < ? \text{ super } U_1, \dots, ? \text{ super } U_l > [S_k = U]$. Then this algorithm is applied recursively to the constraint $A >> V$.
 - ♦ Otherwise, if A is of the form $G < \dots, X_{k-1}, ? \text{ super } W, \dots, X_{k+1}, \dots >$, this algorithm is applied recursively to the constraint $W << U$.
 - ♦ Otherwise, no constraint is implied on T_j .

Next, for each type variable T_j ($1 \leq j \leq n$), the implied equality constraints are resolved as follows.

For each implied equality constraint $T_j = U$ or $U = T_j$:

- If U is not one of the type parameters of the method, then U is the type inferred for T_j . Then all remaining constraints involving T_j are rewritten such that T_j is replaced with U . There are necessarily no further equality constraints involving T_j , and processing continues with the next type parameter, if any.
- Otherwise, if U is T_j , then this constraint carries no information and may be discarded.
- Otherwise, the constraint is of the form $T_j = T_k$ for $j \neq k$. Then all constraints involving T_j are rewritten such that T_j is replaced with T_k , and processing continues with the next type variable.

Then, for each remaining type variable T_j , the constraints $T_j :> U$ are considered. Given that these constraints are $T_j :> U_1 \dots T_j :> U_k$, the type of T_j is inferred as $\text{lub}(U_1 \dots U_k)$, computed as follows:

For a type U , we write $\text{ST}(U)$ for the set of supertypes of U , and define the erased supertype set of U :

$$\text{EST}(U) = \{ V \mid W \text{ in } \text{ST}(U) \text{ and } V = |W| \} \text{ where } |W| \text{ is the erasure of } W.$$

The erased candidate set for type parameter T_j , EC, is the intersection of all the sets $\text{EST}(U)$ for each U in $U_1 \dots U_k$.

The minimal erased candidate set for T_j is:

$$\text{MEC} = \{ V \mid V \text{ in } \text{EC}, \text{ and for all } W \neq V \text{ in } \text{EC}, \text{ it is not the case that } W <: V \}$$

For any element G of MEC that is a generic type declaration, define the relevant invocations of G , $\text{Inv}(G)$, to be:

$$\text{Inv}(G) = \{ V \mid 1 \leq i \leq k: V \text{ in } \text{ST}(U_i), V = G\langle \dots \rangle \}$$

Define $\text{CandidateInvocation}(G) = \text{lci}(\text{Inv}(G))$, where lci , the least containing invocation, is defined:

- $\text{lci}(S) = \text{lci}(e_1, \dots, e_n)$ where e_i ($1 \leq i \leq n$) in S
- $\text{lci}(e_1, \dots, e_n) = \text{lci}(\text{lci}(e_1, e_2), e_3, \dots, e_n)$
- $\text{lci}(G\langle X_1, \dots, X_n \rangle, G\langle Y_1, \dots, Y_n \rangle) = G\langle \text{lcta}(X_1, Y_1), \dots, \text{lcta}(X_n, Y_n) \rangle$
- $\text{lci}(G\langle X_1, \dots, X_n \rangle) = G\langle \text{lcta}(X_1), \dots, \text{lcta}(X_n) \rangle$

where $\text{lcta}()$ is the the least containing type argument function defined (assuming U and V are type expressions) as:

- $\text{lcta}(U, V) = U$ if $U = V$, otherwise $? \text{ extends } \text{lub}(U, V)$
- $\text{lcta}(U, ? \text{ extends } V) = ? \text{ extends } \text{lub}(U, V)$
- $\text{lcta}(U, ? \text{ super } V) = ? \text{ super } \text{glb}(U, V)$
- $\text{lcta}(? \text{ extends } U, ? \text{ extends } V) = ? \text{ extends } \text{lub}(U, V)$
- $\text{lcta}(? \text{ extends } U, ? \text{ super } V) = U$ if $U = V$, otherwise $?$
- $\text{lcta}(? \text{ super } U, ? \text{ super } V) = ? \text{ super } \text{glb}(U, V)$
- $\text{lcta}(U) = ?$ if U 's upper bound is `Object`, otherwise $? \text{ extends } \text{lub}(U, \text{Object})$

where $\text{glb}()$ is as defined in (§5.1.10).

Define $\text{Candidate}(W) = \text{CandidateInvocation}(W)$ if W is generic, W otherwise.

The inferred type for T_j , $\text{lub}(U_1 \dots U_k)$, is $\text{Candidate}(W_1) \& \dots \& \text{Candidate}(W_r)$, where W_i ($1 \leq i \leq r$) are the elements of MEC.

It is possible that the process above yields an infinite type. This is permissible, and a Java compiler must recognize such situations and represent them appropriately using cyclic data structures.

15.12.2.8 Inferring Unresolved Type Arguments

If any of the method's type arguments were not inferred from the types of the actual arguments, they are now inferred as follows.

First, if a type parameter T has been inferred as type C , and T 's bound includes an uninferred type variable X , then X may be inferred by unifying T 's bound with C 's type arguments. Then:

- If the method result occurs in a context where it will be subject to assignment conversion (§5.2) to a type S , then let R be the declared result type of the method, and let $R' = R[T_1=B(T_1) \dots T_n=B(T_n)]$, where $B(T_i)$ is the type inferred for T_i in the previous section or T_i if no type was inferred.

If S is a reference type, then let S' be S . Otherwise, if S is a primitive type, then let S' be the result of applying boxing conversion (§5.1.7) to S .

Then, a set of initial constraints consisting of:

- ♦ the constraint $S' >> R'$, provided R is not `void`; and
- ♦ additional constraints $B_i[T_1=B(T_1) \dots T_n=B(T_n)] >> T_i$, where B_i is the declared bound of T_i ,
- ♦ additional constraints $B(T_i) << B_i[T_1=B(T_1) \dots T_n=B(T_n)]$, where B_i is the declared bound of T_i ,
- ♦ for any constraint of the form $V >> T_i$ generated in §15.12.2.7: a constraint $V[T_1=B(T_1) \dots T_n=B(T_n)] >> T_i$.
- ♦ for any constraint of the form $T_i = V$ generated in §15.12.2.7: a constraint $T_i = V[T_1=B(T_1) \dots T_n=B(T_n)]$.

is created and used to infer constraints on the type arguments using the algorithm of §15.12.2.7.

Any equality constraints are resolved, and then, for each remaining constraint of the form $T_i <: U_k$, the argument T_i is inferred to be $\text{glb}(U_1, \dots, U_k)$ (§5.1.10).

If T_i appears as a type argument in any U_k , then T_i is inferred to be a type variable X whose upper bound is the parameterized type given by $\text{glb}(U_1[T_i=X], \dots, U_k[T_i=X])$ and whose lower bound is the null type.

Any remaining type variable T that has not yet been inferred is then inferred to have type `Object`. If a previously inferred type variable P uses T , then P is inferred to be $P[T=\text{Object}]$.

- Otherwise, the unresolved type arguments are inferred by invoking the procedure described in this section under the assumption that the method result was assigned to a variable of type `Object`.

15.12.3 Compile-Time Step 3: Is the Chosen Method Appropriate?

If there is a most specific method declaration for a method invocation, it is called the *compile-time declaration* for the method invocation. Further checks must be made on the compile-time declaration:

- If the method invocation has, before the left parenthesis, a *MethodName* of the form *Identifier*, and the method is an instance method, then:
 - ♦ If the invocation appears within a static context (§8.1.3), then a compile-time error occurs. (The reason is that a method invocation of this form cannot be used to invoke an instance method in places where this (§15.8.3) is not defined.)
 - ♦ Otherwise, let *C* be the innermost enclosing class of which the method is a member. If the invocation is not directly enclosed by *C* or an inner class of *C*, then a compile-time error occurs.
- If the method invocation has, before the left parenthesis, a *MethodName* of the form *TypeName* . *Identifier*, or if the method invocation, before the left parenthesis, has the form *TypeName* . *NonWildTypeArguments Identifier*, then the compile-time declaration should be `static`.

If the compile-time declaration for the method invocation is for an instance method, then a compile-time error occurs. (The reason is that a method invocation of this form does not specify a reference to an object that can serve as this within the instance method.)

- If the method invocation has, before the left parenthesis, the form `super` . *NonWildTypeArguments_{opt} Identifier*, then:
 - ♦ If the method is `abstract`, a compile-time error occurs.
 - ♦ If the method invocation occurs in a static context, a compile-time error occurs.
- If the method invocation has, before the left parenthesis, the form *ClassName* . `super` . *NonWildTypeArguments_{opt} Identifier*, then:
 - ♦ If the method is `abstract`, a compile-time error occurs.
 - ♦ If the method invocation occurs in a static context, a compile-time error occurs.
 - ♦ Otherwise, let *C* be the class denoted by *ClassName*. If the invocation is not directly enclosed by *C* or an inner class of *C*, then a compile-time error occurs.
- If the compile-time declaration for the method invocation is `void`, then the method invocation must be a top-level expression, that is, the *Expression* in an expression statement (§14.8) or in the *ForInit* or *ForUpdate* part of a `for`

statement (§14.14), or a compile-time error occurs. (The reason is that such a method invocation produces no value and so must be used only in a situation where a value is not needed.)

The following compile-time information is then associated with the method invocation for use at run time:

- The name of the method.
- The qualifying type of the method invocation (§13.1).
- The number of parameters and the types of the parameters, in order.
- The result type, or `void`.
- The invocation mode, computed as follows:
 - ♦ If the compile-time declaration has the `static` modifier, then the invocation mode is `static`.
 - ♦ Otherwise, if the compile-time declaration has the `private` modifier, then the invocation mode is `nonvirtual`.
 - ♦ Otherwise, if the part of the method invocation before the left parenthesis is of the form `super . Identifier` or of the form `ClassName . super . Identifier`, then the invocation mode is `super`.
 - ♦ Otherwise, if the compile-time declaration is in an interface, then the invocation mode is `interface`.
 - ♦ Otherwise, the invocation mode is `virtual`.

If the compile-time declaration for the method invocation is not `void`, then the type of the method invocation expression is the result type specified in the compile-time declaration.

15.12.4 Runtime Evaluation of Method Invocation

At run time, method invocation requires five steps. First, a *target reference* may be computed. Second, the argument expressions are evaluated. Third, the accessibility of the method to be invoked is checked. Fourth, the actual code for the method to be executed is located. Fifth, a new activation frame is created, synchronization is performed if necessary, and control is transferred to the method code.

15.12.4.1 Compute Target Reference (If Necessary)

There are several cases to consider, depending on which of the five productions for *MethodInvocation* (§15.12) is involved:

1. If the first production for *MethodInvocation*, which includes a *MethodName*, is involved, then there are three subcases:

- If the *MethodName* is a simple name, that is, just an *Identifier*, then there are two subcases:
 - ♦ If the invocation mode is *static*, then there is no target reference.
 - ♦ Otherwise, let *T* be the enclosing type declaration of which the method is a member, and let *n* be an integer such that *T* is the *n*'th lexically enclosing type declaration (§8.1.3) of the class whose declaration immediately contains the method invocation. Then the target reference is the *n*'th lexically enclosing instance (§8.1.3) of *this*.

It is a compile-time error if the *n*'th lexically enclosing instance (§8.1.3) of *this* does not exist.

- If the *MethodName* is a qualified name of the form *TypeName* . *Identifier*, then there is no target reference.
 - If the *MethodName* is a qualified name of the form *FieldName* . *Identifier*, then there are two subcases:
 - ♦ If the invocation mode is *static*, then there is no target reference. The expression *FieldName* is evaluated, but the result is then discarded.
 - ♦ Otherwise, the target reference is the value of the expression *FieldName*.
2. If the second production for *MethodInvocation*, which includes a *Primary*, is involved, then there are two subcases:
 - If the invocation mode is *static*, then there is no target reference. The expression *Primary* is evaluated, but the result is then discarded.
 - Otherwise, the expression *Primary* is evaluated and the result is used as the target reference.

In either case, if the evaluation of the *Primary* expression completes abruptly, then no part of any argument expression appears to have been evaluated, and the method invocation completes abruptly for the same reason.

3. If the third production for *MethodInvocation*, which includes the keyword *super*, is involved, then the target reference is the value of *this*.

4. If the fourth production for *MethodInvocation*, *ClassName* . *super*, is involved, then the target reference is the value of *ClassName.this*.
5. If the fifth production for *MethodInvocation*, beginning with *TypeName* . *NonWildTypeArguments*, is involved, then there is no target reference.

15.12.4.2 Evaluate Arguments

The process of evaluating of the argument list differs, depending on whether the method being invoked is a fixed arity method or a variable arity method (§8.4.1).

If the method being invoked is a variable arity method *m*, it necessarily has $n > 0$ formal parameters. The final formal parameter of *m* necessarily has type *T*[] for some *T*, and *m* is necessarily being invoked with $k \geq 0$ actual argument expressions.

If *m* is being invoked with $k \neq n$ actual argument expressions, or, if *m* is being invoked with $k = n$ actual argument expressions and the type of the *k*'th argument expression is not assignment compatible with *T*[], then the argument list (*e*₁, ..., *e*_{*n*-1}, *e*_{*n*}, ..., *e*_{*k*}) is evaluated as if it were written as (*e*₁, ..., *e*_{*n*-1}, new *T*[] { *e*_{*n*}, ..., *e*_{*k*} }).

The argument expressions (possibly rewritten as described above) are now evaluated to yield *argument values*. Each argument value corresponds to exactly one of the method's *n* formal parameters.

The argument expressions, if any, are evaluated in order, from left to right. If the evaluation of any argument expression completes abruptly, then no part of any argument expression to its right appears to have been evaluated, and the method invocation completes abruptly for the same reason. The result of evaluating the *j*'th argument expression is the *j*'th argument value, for $1 \leq j \leq n$. Evaluation then continues, using the argument values, as described below.

15.12.4.3 Check Accessibility of Type and Method

Let *C* be the class containing the method invocation, and let *T* be the qualifying type of the method invocation (§13.1), and let *m* be the name of the method as determined at compile time (§15.12.3).

An implementation of the Java programming language must ensure, as part of linkage, that the method *m* still exists in the type *T*. If this is not true, then a *NoSuchMethodError* (which is a subclass of *IncompatibleClassChangeError*) occurs.

If the invocation mode is *interface*, then the implementation must also check that the target reference type still implements the specified interface.

If the target reference type does not still implement the interface, then an `IncompatibleClassChangeError` occurs.

The implementation must also ensure, during linkage, that the type *T* and the method *m* are accessible. For the type *T*:

- If *T* is in the same package as *C*, then *T* is accessible.
- If *T* is in a different package than *C*, and *T* is `public`, then *T* is accessible.
- If *T* is in a different package than *C*, and *T* is `protected`, then *T* is accessible if and only if *C* is a subclass of *T*.

For the method *m*:

- If *m* is `public`, then *m* is accessible. (All members of interfaces are `public` (§9.2).)
- If *m* is `protected`, then *m* is accessible if and only if either *T* is in the same package as *C*, or *C* is *T* or a subclass of *T*.
- If *m* has default (package) access, then *m* is accessible if and only if *T* is in the same package as *C*.
- If *m* is `private`, then *m* is accessible if and only if *C* is *T*, or *C* encloses *T*, or *T* encloses *C*, or *T* and *C* are both enclosed by a third class.

If either *T* or *m* is not accessible, then an `IllegalAccessException` occurs (§12.3).

15.12.4.4 *Locate Method to Invoke*

The strategy for method lookup depends on the invocation mode.

If the invocation mode is `static`, no target reference is needed and overriding is not allowed. Method *m* of class *T* is the one to be invoked.

Otherwise, an instance method is to be invoked and there is a target reference. If the target reference is `null`, a `NullPointerException` is thrown at this point. Otherwise, the target reference is said to refer to a *target object* and will be used as the value of the keyword `this` in the invoked method. The other four possibilities for the invocation mode are then considered.

If the invocation mode is `nonvirtual`, overriding is not allowed. Method *m* of class *T* is the one to be invoked.

Otherwise, the invocation mode is `interface`, `virtual`, or `super`, and overriding may occur. A *dynamic method lookup* is used. The dynamic lookup process starts from a class *S*, determined as follows:

- If the invocation mode is `interface` or `virtual`, then S is initially the actual run-time class R of the target object.
- If the invocation mode is `super`, then S is initially the qualifying type (§13.1) of the method invocation.

The dynamic method lookup uses the following procedure to search class S , and then the superclasses of class S , as necessary, for method m .

Let X be the compile-time type of the target reference of the method invocation. Then:

1. If class S contains a declaration for a non-abstract method named m with the same descriptor (same number of parameters, the same parameter types, and the same return type) required by the method invocation as determined at compile time (§15.12.3), then:
 - If the invocation mode is `super` or `interface`, then this is the method to be invoked, and the procedure terminates.
 - If the invocation mode is `virtual`, and the declaration in S overrides (§8.4.8.1) $X.m$, then the method declared in S is the method to be invoked, and the procedure terminates.
 - If the invocation mode is `virtual`, and the declaration in S does not override $X.m$, and moreover $X.m$ is declared abstract, then an `AbstractMethodError` is thrown.
2. Otherwise, if S has a superclass, this same lookup procedure is performed recursively using the direct superclass of S in place of S ; the method to be invoked is the result of the recursive invocation of this lookup procedure.

The above procedure (if it terminates without error) will find a non-abstract, accessible method to invoke, provided that all classes and interfaces in the program have been consistently compiled. However, if this is not the case, then various errors may occur. The specification of the behavior of a Java virtual machine under these circumstances is given by *The Java Virtual Machine Specification*.

15.12.4.5 Create Frame, Synchronize, Transfer Control

A method m in some class S has been identified as the one to be invoked.

Now a new *activation frame* is created, containing the target reference (if any) and the argument values (if any), as well as enough space for the local variables and stack for the method to be invoked and any other bookkeeping information that may be required by the implementation (stack pointer, program counter, reference to

previous activation frame, and the like). If there is not sufficient memory available to create such an activation frame, a `StackOverflowError` is thrown.

The newly created activation frame becomes the current activation frame. The effect of this is to assign the argument values to corresponding freshly created parameter variables of the method, and to make the target reference available as `this`, if there is a target reference. Before each argument value is assigned to its corresponding parameter variable, it is subjected to method invocation conversion (§5.3), which includes any required value set conversion (§5.1.13).

If the erasure of the type of the method being invoked differs in its signature from the erasure of the type of the compile-time declaration for the method invocation (§15.12.3), then if any of the argument values is an object which is not an instance of a subclass or subinterface of the erasure of the corresponding formal parameter type in the compile-time declaration for the method invocation, then a `ClassCastException` is thrown.

If the method *m* is a native method but the necessary native, implementation-dependent binary code has not been loaded or otherwise cannot be dynamically linked, then an `UnsatisfiedLinkError` is thrown.

If the method *m* is not synchronized, control is transferred to the body of the method *m* to be invoked.

If the method *m* is synchronized, then an object must be locked before the transfer of control. No further progress can be made until the current thread can obtain the lock. If there is a target reference, then the target object must be locked; otherwise the `Class` object for class *S*, the class of the method *m*, must be locked. Control is then transferred to the body of the method *m* to be invoked. The object is automatically unlocked when execution of the body of the method has completed, whether normally or abruptly. The locking and unlocking behavior is exactly as if the body of the method were embedded in a `synchronized` statement (§14.19).

15.13 Array Access Expressions

An array access expression refers to a variable that is a component of an array.

ArrayAccess:

ExpressionName [*Expression*]

PrimaryNoNewArray [*Expression*]

An array access expression contains two subexpressions, the *array reference expression* (before the left bracket) and the *index expression* (within the brackets). Note that the array reference expression may be a name or any primary expression that is not an array creation expression (§15.10).

The type of the array reference expression must be an array type (call it $T[]$, an array whose components are of type T), or a compile-time error occurs.

The type of the array access expression is the result of applying capture conversion (§5.1.10) to T .

The index expression undergoes unary numeric promotion (§5.6.1); the promoted type must be `int`.

The result of an array reference is a variable of type T , namely the variable within the array selected by the value of the index expression.

This resulting variable, which is a component of the array, is never considered `final`, even if the array reference was obtained from a `final` variable.

15.13.1 Runtime Evaluation of Array Access

An array access expression is evaluated using the following procedure:

- First, the array reference expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason and the index expression is not evaluated.
- Otherwise, the index expression is evaluated. If this evaluation completes abruptly, then the array access completes abruptly for the same reason.
- Otherwise, if the value of the array reference expression is `null`, then a `NullPointerException` is thrown.
- Otherwise, the value of the array reference expression indeed refers to an array. If the value of the index expression is less than zero, or greater than or equal to the array's `length`, then an `ArrayIndexOutOfBoundsException` is thrown.
- Otherwise, the result of the array access is the variable of type T , within the array, selected by the value of the index expression.

Note that this resulting variable, which is a component of the array, is never considered `final`, even if the array reference expression is a `final` variable.

15.14 Postfix Expressions

Postfix expressions include uses of the postfix ++ and -- operators. Also, as discussed in §15.8, names are not considered to be primary expressions, but are handled separately in the grammar to avoid certain ambiguities. They become interchangeable only here, at the level of precedence of postfix expressions.

PostfixExpression:

Primary

ExpressionName

PostIncrementExpression

PostDecrementExpression

15.14.1 Expression Names

The rules for evaluating expression names are given in §6.5.6.

15.14.2 Postfix Increment Operator ++

PostIncrementExpression:

PostfixExpression ++

A postfix expression followed by a ++ operator is a postfix increment expression.

The result of the postfix expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the postfix increment expression is the type of the variable. The result of the postfix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value 1 is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (§5.6.2) is performed on the value 1 and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of the variable before it is stored. The value of the postfix increment expression is the value of the variable *before* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, value set conversion is applied to the sum prior to its being stored in the variable.

A variable that is declared `final` cannot be incremented because when an access of such a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a postfix increment operator.

15.14.3 Postfix Decrement Operator --

PostDecrementExpression:

PostfixExpression --

A postfix expression followed by a `--` operator is a postfix decrement expression.

The result of the postfix expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the postfix decrement expression is the type of the variable. The result of the postfix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value 1 is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (§5.6.2) is performed on the value 1 and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of the variable before it is stored. The value of the postfix decrement expression is the value of the variable *before* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, value set conversion is applied to the difference prior to its being stored in the variable.

A variable that is declared `final` cannot be decremented because when an access of such a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a postfix decrement operator.

15.15 Unary Operators

The *unary operators* include `+`, `-`, `++`, `--`, `~`, `!`, and cast operators.

Expressions with unary operators group right-to-left, so that `--x` means the same as `-(~x)`.

UnaryExpression:

PreIncrementExpression

PreDecrementExpression

+ UnaryExpression

- UnaryExpression

UnaryExpressionNotPlusMinus

PreIncrementExpression:

++ UnaryExpression

PreDecrementExpression:

-- UnaryExpression

UnaryExpressionNotPlusMinus:

PostfixExpression

~ UnaryExpression

! UnaryExpression

CastExpression

15.15.1 Prefix Increment Operator ++

A unary expression preceded by a ++ operator is a prefix increment expression.

The result of the unary expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the prefix increment expression is the type of the variable. The result of the prefix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value 1 is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (§5.6.2) is performed on the value 1 and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of the variable before it is stored. The value of the prefix increment expression is the value of the variable *after* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, value set conversion is applied to the sum prior to its being stored in the variable.

A variable that is declared `final` cannot be incremented because when an access of such a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a prefix increment operator.

15.15.2 Prefix Decrement Operator --

A unary expression preceded by a -- operator is a prefix decrement expression.

The result of the unary expression must be a variable of a type that is convertible (§5.1.8) to a numeric type, or a compile-time error occurs.

The type of the prefix decrement expression is the type of the variable. The result of the prefix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value 1 is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (§5.6.2) is performed on the value 1 and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (§5.1.3) and/or subjected to boxing conversion (§5.1.7) to the type of the variable before it is stored. The value of the prefix decrement expression is the value of the variable *after* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (§5.1.8) and value set conversion (§5.1.13). If necessary, format conversion is applied to the difference prior to its being stored in the variable.

A variable that is declared `final` cannot be decremented because when an access of such a `final` variable is used as an expression, the result is a value, not a variable. Thus, it cannot be used as the operand of a prefix decrement operator.

15.15.3 Unary Plus Operator +

The type of the operand expression of the unary + operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

Unary numeric promotion (§5.6.1) is performed on the operand. The type of the unary plus expression is the promoted type of the operand. The result of the unary plus expression is not a variable, but a value, even if the result of the operand expression is a variable.

At run time, the value of the unary plus expression is the promoted value of the operand.

15.15.4 Unary Minus Operator -

The type of the operand expression of the unary - operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

Unary numeric promotion (§5.6.1) is performed on the operand. The type of the unary minus expression is the promoted type of the operand.

Note that unary numeric promotion performs value set conversion (§5.1.13). Whatever value set the promoted operand value is drawn from, the unary negation operation is carried out and the result is drawn from that same value set. That result is then subject to further value set conversion.

At run time, the value of the unary minus expression is the arithmetic negation of the promoted value of the operand.

For integer values, negation is the same as subtraction from zero. The Java programming language uses two's-complement representation for integers, and the range of two's-complement values is not symmetric, so negation of the maximum negative `int` or `long` results in that same maximum negative number. Overflow occurs in this case, but no exception is thrown. For all integer values `x`, `-x` equals `(~x)+1`.

For floating-point values, negation is *not* the same as subtraction from zero, because if `x` is `+0.0`, then `0.0-x` is `+0.0`, but `-x` is `-0.0`. Unary minus merely inverts the sign of a floating-point number. Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.

15.15.5 Bitwise Complement Operator ~

The type of the operand expression of the unary ~ operator must be a type that is convertible (§5.1.8) to a primitive integral type, or a compile-time error occurs.

Unary numeric promotion (§5.6.1) is performed on the operand. The type of the unary bitwise complement expression is the promoted type of the operand.

At run time, the value of the unary bitwise complement expression is the bitwise complement of the promoted value of the operand; note that, in all cases, `~x` equals `(-x)-1`.

15.15.6 Logical Complement Operator !

The type of the operand expression of the unary ! operator must be `boolean` or `Boolean`, or a compile-time error occurs.

The type of the unary logical complement expression is `boolean`.

At run time, the operand is subject to unboxing conversion (§5.1.8) if necessary; the value of the unary logical complement expression is `true` if the (possibly converted) operand value is `false`, and `false` if the (possibly converted) operand value is `true`.

15.16 Cast Expressions

A cast expression converts, at run time, a value of one numeric type to a similar value of another numeric type; or confirms, at compile time, that the type of an expression is `boolean`; or checks, at run time, that a reference value refers to an object whose class is compatible with a specified reference type.

CastExpression:

(*PrimitiveType*) *UnaryExpression*

(*ReferenceType*) *UnaryExpressionNotPlusMinus*

The type of a cast expression is the result of applying capture conversion (§5.1.10) to the type whose name appears within the parentheses. (The parentheses and the type they contain are sometimes called the *cast operator*.)

The result of a cast expression is not a variable, but a value, even if the result of the operand expression is a variable.

A cast operator has no effect on the choice of value set (§4.2.3) for a value of type `float` or type `double`. Consequently, a cast to type `float` within an expression that is not FP-strict (§15.4) does not necessarily cause its value to be converted to an element of the float value set, and a cast to type `double` within an expression that is not FP-strict does not necessarily cause its value to be converted to an element of the double value set.

It is a compile-time error if the compile-time type of the operand may never be cast to the type specified by the cast operator according to the rules of casting conversion (§5.5). Otherwise, at run-time, the operand value is converted (if necessary) by casting conversion to the type specified by the cast operator.

A `ClassCastException` is thrown if a cast is found at run time to be impermissible.

15.17 Multiplicative Operators

The operators `*`, `/`, and `%` are called the *multiplicative operators*. They have the same precedence and are syntactically left-associative (they group left-to-right).

MultiplicativeExpression:

UnaryExpression

MultiplicativeExpression `*` *UnaryExpression*

MultiplicativeExpression `/` *UnaryExpression*

MultiplicativeExpression `%` *UnaryExpression*

The type of each of the operands of a multiplicative operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

Binary numeric promotion is performed on the operands (§5.6.2).

The type of a multiplicative expression is the promoted type of its operands.

If the promoted type is `int` or `long`, then integer arithmetic is performed.

If the promoted type is `float` or `double`, then floating-point arithmetic is performed.

15.17.1 Multiplication Operator `*`

The binary `*` operator performs multiplication, producing the product of its operands.

Multiplication is a commutative operation if the operand expressions have no side effects.

Integer multiplication is associative when the operands are all of the same type, but floating-point multiplication is not associative.

If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format. As a result, if overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two operand values.

The result of a floating-point multiplication is determined by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result is positive if both operands have the same sign, and negative if the operands have different signs.

- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity. The sign is determined by the rule stated above.
- In the remaining cases, where neither an infinity nor NaN is involved, the exact mathematical product is computed. A floating-point value set is then chosen:
 - ♦ If the multiplication expression is FP-strict (§15.4):
 - ✧ If the type of the multiplication expression is `float`, then the float value set must be chosen.
 - ✧ If the type of the multiplication expression is `double`, then the double value set must be chosen.
 - ♦ If the multiplication expression is not FP-strict:
 - ✧ If the type of the multiplication expression is `float`, then either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
 - ✧ If the type of the multiplication expression is `double`, then either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

Next, a value must be chosen from the chosen value set to represent the product.

If the magnitude of the product is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign.

Otherwise, the product is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java programming language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

Despite the fact that overflow, underflow, or loss of information may occur, evaluation of a multiplication operator `*` never throws a run-time exception.

15.17.2 Division Operator /

The binary `/` operator performs division, producing the quotient of its operands. The left-hand operand is the *dividend* and the right-hand operand is the *divisor*.

Integer division rounds toward 0. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$. Moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is `-1`, then integer overflow occurs and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is `0`, then an `ArithmeticException` is thrown.

The result of a floating-point division is determined by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result is positive if both operands have the same sign, and negative if the operands have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity. The sign is determined by the rule stated above.
- Division of a finite value by an infinity results in a signed zero. The sign is determined by the rule stated above.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero. The sign is determined by the rule stated above.
- Division of a nonzero finite value by a zero results in a signed infinity. The sign is determined by the rule stated above.
- In the remaining cases, where neither an infinity nor NaN is involved, the exact mathematical quotient is computed. A floating-point value set is then chosen:
 - ♦ If the division expression is FP-strict (§15.4):
 - ❖ If the type of the division expression is `float`, then the float value set must be chosen.
 - ❖ If the type of the division expression is `double`, then the double value set must be chosen.
 - ♦ If the division expression is not FP-strict:
 - ❖ If the type of the division expression is `float`, then either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
 - ❖ If the type of the division expression is `double`, then either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

Next, a value must be chosen from the chosen value set to represent the quotient.

If the magnitude of the quotient is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign.

Otherwise, the quotient is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java programming language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

Despite the fact that overflow, underflow, division by zero, or loss of information may occur, evaluation of a floating-point division operator / never throws a run-time exception.

15.17.3 Remainder Operator %

The binary % operator is said to yield the remainder of its operands from an implied division; the left-hand operand is the *dividend* and the right-hand operand is the *divisor*.

In C and C++, the remainder operator accepts only integral operands, but in the Java programming language, it also accepts floating-point operands.

The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that $(a/b)*b+(a\%b)$ is equal to a .

If the value of the divisor for an integer remainder operator is 0, then an `ArithmeticException` is thrown.

The result of a floating-point remainder operation as computed by the % operator is *not* the same as that produced by the remainder operation defined by IEEE 754. The IEEE 754 remainder operation computes the remainder from a rounding division, not a truncating division, and so its behavior is *not* analogous to that of the usual integer remainder operator. Instead, the Java programming language defines % on floating-point operations to behave in a manner analogous to that of the integer remainder operator; this may be compared with the C library function `fmod`. The IEEE 754 remainder operation may be computed by the library routine `Math.IEEEremainder`.

The result of a floating-point remainder operation is determined by the rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- If the result is not NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity, or the divisor is a zero, or both, the result is NaN.

- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
- If the dividend is a zero and the divisor is finite, the result equals the dividend.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the floating-point remainder r from the division of a dividend n by a divisor d is defined by the mathematical relation $r = n - (d \cdot q)$ where q is an integer that is negative only if n/d is negative and positive only if n/d is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of n and d .

Evaluation of a floating-point remainder operator `%` never throws a run-time exception, even if the right-hand operand is zero. Overflow, underflow, or loss of precision cannot occur.

15.18 Additive Operators

The operators `+` and `-` are called the additive operators. They have the same precedence and are syntactically left-associative (they group left-to-right).

AdditiveExpression:

MultiplicativeExpression

AdditiveExpression + *MultiplicativeExpression*

AdditiveExpression - *MultiplicativeExpression*

If the type of either operand of a `+` operator is `String`, then the operation is string concatenation.

Otherwise, the type of each of the operands of the `+` operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

In every case, the type of each of the operands of the binary `-` operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

15.18.1 String Concatenation Operator `+`

If only one operand expression is of type `String`, then string conversion (§5.1.11) is performed on the other operand to produce a string at run time. The result is a reference to a `String` object (newly created, unless the expression is a compile-time constant expression (§15.28)) that is the concatenation of the two operand

strings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string.

If an operand of type `String` is `null`, then the string `"null"` is used instead of that operand.

15.18.2 Additive Operators (+ and -) for Numeric Types

The binary `+` operator performs addition when applied to two operands of numeric type, producing the sum of the operands.

The binary `-` operator performs subtraction, producing the difference of two numeric operands.

Binary numeric promotion is performed on the operands (§5.6.2).

The type of an additive expression on numeric operands is the promoted type of its operands:

- If this promoted type is `int` or `long`, then integer arithmetic is performed.
- If this promoted type is `float` or `double`, then floating-point arithmetic is performed.

Addition is a commutative operation if the operand expressions have no side effects.

Integer addition is associative when the operands are all of the same type, but floating-point addition is not associative.

If an integer addition overflows, then the result is the low-order bits of the mathematical sum as represented in some sufficiently large two's-complement format. If overflow occurs, then the sign of the result is not the same as the sign of the mathematical sum of the two operand values.

The result of a floating-point addition is determined using the following rules of IEEE 754 arithmetic:

- If either operand is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and a finite value is equal to the infinite operand.
- The sum of two zeros of opposite sign is positive zero.
- The sum of two zeros of the same sign is the zero of that sign.

- The sum of a zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the exact mathematical sum is computed. A floating-point value set is then chosen:
 - ♦ If the addition expression is FP-strict (§15.4):
 - ❖ If the type of the addition expression is `float`, then the float value set must be chosen.
 - ❖ If the type of the addition expression is `double`, then the double value set must be chosen.
 - ♦ If the addition expression is not FP-strict:
 - ❖ If the type of the addition expression is `float`, then either the float value set or the float-extended-exponent value set may be chosen, at the whim of the implementation.
 - ❖ If the type of the addition expression is `double`, then either the double value set or the double-extended-exponent value set may be chosen, at the whim of the implementation.

Next, a value must be chosen from the chosen value set to represent the sum.

If the magnitude of the sum is too large to represent, we say the operation overflows; the result is then an infinity of appropriate sign.

Otherwise, the sum is rounded to the nearest value in the chosen value set using IEEE 754 round-to-nearest mode. The Java programming language requires support of gradual underflow as defined by IEEE 754 (§4.2.4).

The binary `-` operator performs subtraction when applied to two operands of numeric type, producing the difference of its operands; the left-hand operand is the *minuend* and the right-hand operand is the *subtrahend*.

For both integer and floating-point subtraction, it is always the case that `a-b` produces the same result as `a+(-b)`.

Note that, for integer values, subtraction from zero is the same as negation. However, for floating-point operands, subtraction from zero is *not* the same as negation, because if `x` is `+0.0`, then `0.0-x` is `+0.0`, but `-x` is `-0.0`.

Despite the fact that overflow, underflow, or loss of information may occur, evaluation of a numeric additive operator never throws a run-time exception.

15.19 Shift Operators

The shift operators include left shift `<<`, signed right shift `>>`, and unsigned right shift `>>>`; they are syntactically left-associative (they group left-to-right). The left-hand operand of a shift operator is the value to be shifted; the right-hand operand specifies the shift distance.

ShiftExpression:
AdditiveExpression
ShiftExpression `<<` *AdditiveExpression*
ShiftExpression `>>` *AdditiveExpression*
ShiftExpression `>>>` *AdditiveExpression*

The type of each of the operands of a shift operator must be a type that is convertible (§5.1.8) to a primitive integral type, or a compile-time error occurs.

Binary numeric promotion (§5.6.2) is *not* performed on the operands; rather, unary numeric promotion (§5.6.1) is performed on each operand separately.

The type of the shift expression is the promoted type of the left-hand operand.

If the promoted type of the left-hand operand is `int`, only the five lowest-order bits of the right-hand operand are used as the shift distance.

It is as if the right-hand operand were subjected to a bitwise logical AND operator `&` (§15.22.1) with the mask value `0x1f`. The shift distance actually used is therefore always in the range 0 to 31, inclusive.

If the promoted type of the left-hand operand is `long`, then only the six lowest-order bits of the right-hand operand are used as the shift distance.

It is as if the right-hand operand were subjected to a bitwise logical AND operator `&` (§15.22.1) with the mask value `0x3f`. The shift distance actually used is therefore always in the range 0 to 63, inclusive.

At run time, shift operations are performed on the two's-complement integer representation of the value of the left operand.

The value of `n << s` is `n` left-shifted `s` bit positions; this is equivalent (even if overflow occurs) to multiplication by two to the power `s`.

The value of `n >> s` is `n` right-shifted `s` bit positions with sign-extension. The resulting value is $\lfloor n / 2^s \rfloor$. For non-negative values of `n`, this is equivalent to truncating integer division, as computed by the integer division operator `/`, by two to the power `s`.

The value of $n \ggg s$ is n right-shifted s bit positions with zero-extension.

- If n is positive, then the result is the same as that of $n \gg s$.
- If n is negative and the type of the left-hand operand is `int`, then the result is equal to that of the expression $(n \gg s) + (2 \ll \sim s)$.
- If n is negative and the type of the left-hand operand is `long`, then the result is equal to that of the expression $(n \gg s) + (2L \ll \sim s)$.

15.20 Relational Operators

The *relational operators* are syntactically left-associative (they group left-to-right), but this fact is not useful. For example, $a < b < c$ parses as $(a < b) < c$, which is always a compile-time error, because the type of $a < b$ is always `boolean` and $<$ is not an operator on `boolean` values.

RelationalExpression:

ShiftExpression

RelationalExpression $<$ *ShiftExpression*

RelationalExpression $>$ *ShiftExpression*

RelationalExpression $<=$ *ShiftExpression*

RelationalExpression $>=$ *ShiftExpression*

RelationalExpression instanceof *ReferenceType*

The type of a relational expression is always `boolean`.

15.20.1 Numerical Comparison Operators $<$, $<=$, $>$, and $>=$

The type of each of the operands of a numerical comparison operator must be a type that is convertible (§5.1.8) to a primitive numeric type, or a compile-time error occurs.

Binary numeric promotion is performed on the operands (§5.6.2).

If the promoted type of the operands is `int` or `long`, then signed integer comparison is performed.

If the promoted type is `float` or `double`, then floating-point comparison is performed.

Comparison is carried out accurately on floating-point values, no matter what value sets their representing values were drawn from.

The result of a floating-point comparison, as determined by the specification of the IEEE 754 standard, is:

- If either operand is NaN, then the result is `false`.
- All values other than NaN are ordered, with negative infinity less than all finite values, and positive infinity greater than all finite values.
- Positive zero and negative zero are considered equal.

Subject to these considerations for floating-point numbers, the following rules then hold for integer operands or for floating-point operands other than NaN:

- The value produced by the `<` operator is `true` if the value of the left-hand operand is less than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `<=` operator is `true` if the value of the left-hand operand is less than or equal to the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `>` operator is `true` if the value of the left-hand operand is greater than the value of the right-hand operand, and otherwise is `false`.
- The value produced by the `>=` operator is `true` if the value of the left-hand operand is greater than or equal to the value of the right-hand operand, and otherwise is `false`.

15.20.2 Type Comparison Operator `instanceof`

The type of a *RelationalExpression* operand of the `instanceof` operator must be a reference type or the null type; otherwise, a compile-time error occurs.

It is a compile-time error if the *ReferenceType* mentioned after the `instanceof` operator does not denote a reference type that is reifiable (§4.7).

At run time, the result of the `instanceof` operator is `true` if the value of the *RelationalExpression* is not `null` and the reference could be cast (§15.16) to the *ReferenceType* without raising a `ClassCastException`. Otherwise the result is `false`.

If a cast of the *RelationalExpression* to the *ReferenceType* would be rejected as a compile-time error, then the `instanceof` relational expression likewise produces a compile-time error. In such a situation, the result of the `instanceof` expression could never be `true`.

15.21 Equality Operators

The equality operators are syntactically left-associative (they group left-to-right), but this fact is essentially never useful. For example, `a==b==c` parses as `(a==b)==c`. The result type of `a==b` is always `boolean`, and `c` must therefore be of type `boolean` or a compile-time error occurs. Thus, `a==b==c` does not test to see whether `a`, `b`, and `c` are all equal.

EqualityExpression:

RelationalExpression

EqualityExpression == *RelationalExpression*

EqualityExpression != *RelationalExpression*

The `==` (equal to) and the `!=` (not equal to) operators are analogous to the relational operators except for their lower precedence. Thus, `a<b==c<d` is `true` whenever `a<b` and `c<d` have the same truth value.

The equality operators may be used to compare two operands that are convertible (§5.1.8) to numeric type, or two operands of type `boolean` or `Boolean`, or two operands that are each of either reference type or the null type. All other cases result in a compile-time error.

The type of an equality expression is always `boolean`.

In all cases, `a!=b` produces the same result as `!(a==b)`.

The equality operators are commutative if the operand expressions have no side effects.

15.21.1 Numerical Equality Operators `==` and `!=`

If the operands of an equality operator are both of numeric type, or one is of numeric type and the other is convertible (§5.1.8) to numeric type, binary numeric promotion is performed on the operands (§5.6.2).

If the promoted type of the operands is `int` or `long`, then an integer equality test is performed.

If the promoted type is `float` or `double`, then a floating-point equality test is performed.

Comparison is carried out accurately on floating-point values, no matter what value sets their representing values were drawn from.

Floating-point equality testing is performed in accordance with the rules of the IEEE 754 standard:

- If either operand is NaN, then the result of == is `false` but the result of != is `true`.
Indeed, the test `x!=x` is `true` if and only if the value of `x` is NaN.
- Positive zero and negative zero are considered equal.
- Otherwise, two distinct floating-point values are considered unequal by the equality operators.

In particular, there is one value representing positive infinity and one value representing negative infinity; each compares equal only to itself, and each compares unequal to all other values.

Subject to these considerations for floating-point numbers, the following rules then hold for integer operands or for floating-point operands other than NaN:

- The value produced by the == operator is `true` if the value of the left-hand operand is equal to the value of the right-hand operand; otherwise, the result is `false`.
- The value produced by the != operator is `true` if the value of the left-hand operand is not equal to the value of the right-hand operand; otherwise, the result is `false`.

15.21.2 Boolean Equality Operators == and !=

If the operands of an equality operator are both of type `boolean`, or if one operand is of type `boolean` and the other is of type `Boolean`, then the operation is boolean equality.

The boolean equality operators are associative.

If one of the operands is of type `Boolean`, it is subjected to unboxing conversion (§5.1.8).

The result of == is `true` if the operands (after any required unboxing conversion) are both `true` or both `false`; otherwise, the result is `false`.

The result of != is `false` if the operands are both `true` or both `false`; otherwise, the result is `true`.

15.21.3 Reference Equality Operators == and !=

If the operands of an equality operator are both of either reference type or the null type, then the operation is object equality.

A compile-time error occurs if it is impossible to convert the type of either operand to the type of the other by a casting conversion (§5.5). The run-time values of the two operands would necessarily be unequal.

At run time, the result of == is true if the operand values are both null or both refer to the same object or array; otherwise, the result is false.

The result of != is false if the operand values are both null or both refer to the same object or array; otherwise, the result is true.

While == may be used to compare references of type String, such an equality test determines whether or not the two operands refer to the same String object. The result is false if the operands are distinct String objects, even if they contain the same sequence of characters. The contents of two strings s and t can be tested for equality by the method invocation s.equals(t). See also §3.10.5.

15.22 Bitwise and Logical Operators

The *bitwise operators* and *logical operators* include the AND operator &, exclusive OR operator ^, and inclusive OR operator |. These operators have different precedence, with & having the highest precedence and | the lowest precedence.

Each of these operators is syntactically left-associative (each groups left-to-right).

Each operator is commutative if the operand expressions have no side effects.

Each operator is associative.

AndExpression:

EqualityExpression

AndExpression & EqualityExpression

ExclusiveOrExpression:

AndExpression

ExclusiveOrExpression ^ AndExpression

InclusiveOrExpression:

ExclusiveOrExpression

InclusiveOrExpression | ExclusiveOrExpression

The bitwise and logical operators may be used to compare two operands of numeric type or two operands of type `boolean`. All other cases result in a compile-time error.

15.22.1 Integer Bitwise Operators &, ^, and |

When both operands of an operator `&`, `^`, or `|` are of a type that is convertible (§5.1.8) to a primitive integral type, binary numeric promotion is first performed on the operands (§5.6.2).

The type of the bitwise operator expression is the promoted type of the operands.

For `&`, the result value is the bitwise AND of the operand values.

For `^`, the result value is the bitwise exclusive OR of the operand values.

For `|`, the result value is the bitwise inclusive OR of the operand values.

15.22.2 Boolean Logical Operators &, ^, and |

When both operands of a `&`, `^`, or `|` operator are of type `boolean` or `Boolean`, then the type of the bitwise operator expression is `boolean`. In all cases, the operands are subject to unboxing conversion (§5.1.8) as necessary.

For `&`, the result value is `true` if both operand values are `true`; otherwise, the result is `false`.

For `^`, the result value is `true` if the operand values are different; otherwise, the result is `false`.

For `|`, the result value is `false` if both operand values are `false`; otherwise, the result is `true`.

15.23 Conditional-And Operator &&

The && operator is like & (§15.22.2), but evaluates its right-hand operand only if the value of its left-hand operand is `true`.

It is syntactically left-associative (it groups left-to-right).

It is fully associative with respect to both side effects and result value; that is, for any expressions *a*, *b*, and *c*, evaluation of the expression `((a) && (b)) && (c)` produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `(a) && ((b) && (c))`.

ConditionalAndExpression:

InclusiveOrExpression

ConditionalAndExpression && InclusiveOrExpression

Each operand of && must be of type `boolean` or `Boolean`, or a compile-time error occurs.

The type of a conditional-and expression is always `boolean`.

At run time, the left-hand operand expression is evaluated first; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8).

If the resulting value is `false`, the value of the conditional-and expression is `false` and the right-hand operand expression is not evaluated.

If the value of the left-hand operand is `true`, then the right-hand expression is evaluated; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8). The resulting value becomes the value of the conditional-and expression.

Thus, && computes the same result as & on `boolean` operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

15.24 Conditional-Or Operator ||

The || operator is like | (§15.22.2), but evaluates its right-hand operand only if the value of its left-hand operand is `false`.

It is syntactically left-associative (it groups left-to-right).

It is fully associative with respect to both side effects and result value; that is, for any expressions *a*, *b*, and *c*, evaluation of the expression `((a) || (b)) || (c)`

produces the same result, with the same side effects occurring in the same order, as evaluation of the expression `(a) || ((b) || (c))`.

ConditionalOrExpression:

ConditionalAndExpression

ConditionalOrExpression || ConditionalAndExpression

Each operand of `||` must be of type `boolean` or `Boolean`, or a compile-time error occurs.

The type of a conditional-or expression is always `boolean`.

At run time, the left-hand operand expression is evaluated first; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8).

If the resulting value is `true`, the value of the conditional-or expression is `true` and the right-hand operand expression is not evaluated.

If the value of the left-hand operand is `false`, then the right-hand expression is evaluated; if the result has type `Boolean`, it is subjected to unboxing conversion (§5.1.8). The resulting value becomes the value of the conditional-or expression.

Thus, `||` computes the same result as `|` on `boolean` or `Boolean` operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

15.25 Conditional Operator ? :

The conditional operator `? :` uses the boolean value of one expression to decide which of two other expressions should be evaluated.

The conditional operator is syntactically right-associative (it groups right-to-left), so that `a?b:c?d:e?f:g` means the same as `a?b:(c?d:(e?f:g))`.

ConditionalExpression:

ConditionalOrExpression

ConditionalOrExpression ? Expression : ConditionalExpression

The conditional operator has three operand expressions; `?` appears between the first and second expressions, and `:` appears between the second and third expressions.

The first expression must be of type `boolean` or `Boolean`, or a compile-time error occurs.

It is a compile-time error for either the second or the third operand expression to be an invocation of a `void` method.

The type of a conditional expression is determined as follows:

- If the second and third operands have the same type (which may be the null type), then that is the type of the conditional expression.
- If one of the second and third operands is of primitive type T , and the type of the other is the result of applying boxing conversion (§5.1.7) to T , then the type of the conditional expression is T .
- If one of the second and third operands is of the null type and the type of the other is a reference type, then the type of the conditional expression is that reference type.
- Otherwise, if the second and third operands have types that are convertible (§5.1.8) to numeric types, then there are several cases:
 - ♦ If one of the operands is of type `byte` or `Byte` and the other is of type `short` or `Short`, then the type of the conditional expression is `short`.
 - ♦ If one of the operands is of type T where T is `byte`, `short`, or `char`, and the other operand is a constant expression of type `int` whose value is representable in type T , then the type of the conditional expression is T .
 - ♦ If one of the operands is of type T , where T is `Byte`, `Short`, or `Character`, and the other operand is a constant expression of type `int` whose value is representable in the type U which is the result of applying unboxing conversion to T , then the type of the conditional expression is U .
 - ♦ Otherwise, binary numeric promotion (§5.6.2) is applied to the operand types, and the type of the conditional expression is the promoted type of the second and third operands.
- Otherwise, the second and third operands are of types S_1 and S_2 respectively. Let T_1 be the type that results from applying boxing conversion to S_1 , and let T_2 be the type that results from applying boxing conversion to S_2 .

The type of the conditional expression is the result of applying capture conversion (§5.1.10) to $\text{lub}(T_1, T_2)$ (§15.12.2.7).

At run time, the first operand expression of the conditional expression is evaluated first; if necessary, unboxing conversion is performed on the result. The resulting `boolean` value is then used to choose either the second or the third operand expression:

- If the value of the first operand is `true`, then the second operand expression is chosen.
- If the value of the first operand is `false`, then the third operand expression is chosen.

The chosen operand expression is then evaluated and the resulting value is converted to the type of the conditional expression as determined by the rules stated above.

This conversion may include boxing (§5.1.7) or unboxing (§5.1.8) conversion.

The operand expression not chosen is not evaluated for that particular evaluation of the conditional expression.

15.26 Assignment Operators

There are 12 *assignment operators*; all are syntactically right-associative (they group right-to-left). Thus, `a=b=c` means `a=(b=c)`, which assigns the value of `c` to `b` and then assigns the value of `b` to `a`.

AssignmentExpression:
ConditionalExpression
Assignment

Assignment:
LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:
ExpressionName
FieldAccess
ArrayAccess

AssignmentOperator: one of
`= *= /= %= += -= <=> >>= &= ^= |=`

The result of the first operand of an assignment operator must be a variable, or a compile-time error occurs.

This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access (§15.11) or an array access (§15.13).

The type of the assignment expression is the type of the variable after capture conversion (§5.1.10).

At run time, the result of the assignment expression is the value of the variable after the assignment has occurred. The result of an assignment expression is not itself a variable.

A variable that is declared `final` cannot be assigned to (unless it is definitely unassigned (Chapter 16, *Definite Assignment*)), because when an access of such a `final` variable is used as an expression, the result is a value, not a variable, and so it cannot be used as the first operand of an assignment operator.

15.26.1 Simple Assignment Operator =

A compile-time error occurs if the type of the right-hand operand cannot be converted to the type of the variable by assignment conversion (§5.2).

At run time, the expression is evaluated in one of three ways.

If the left-hand operand expression is a field access expression (§15.11) `e.f`, possibly enclosed in one or more pairs of parentheses, then:

- First, the expression `e` is evaluated. If evaluation of `e` completes abruptly, the assignment expression completes abruptly for the same reason.
- Next, the right hand operand is evaluated. If evaluation of the right hand expression completes abruptly, the assignment expression completes abruptly for the same reason.
- Then, if the field denoted by `e.f` is not `static` and the result of the evaluation of `e` above is `null`, then a `NullPointerException` is thrown.
- Otherwise, the variable denoted by `e.f` is assigned the value of the right hand operand as computed above.

If the left-hand operand is an array access expression (§15.13), possibly enclosed in one or more pairs of parentheses, then:

- First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the index subexpression (of the left-hand operand array access expression) and the right-hand operand are not evaluated and no assignment occurs.
- Otherwise, the index subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the

assignment expression completes abruptly for the same reason and the right-hand operand is not evaluated and no assignment occurs.

- Otherwise, the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, if the value of the array reference subexpression is `null`, then no assignment occurs and a `NullPointerException` is thrown.
- Otherwise, the value of the array reference subexpression indeed refers to an array. If the value of the index subexpression is less than zero, or greater than or equal to the `length` of the array, then no assignment occurs and an `ArrayIndexOutOfBoundsException` is thrown.
- Otherwise, the value of the index subexpression is used to select a component of the array referred to by the value of the array reference subexpression.

This component is a variable; call its type *SC*. Also, let *TC* be the type of the left-hand operand of the assignment operator as determined at compile time. Then there are two possibilities:

- ♦ If *TC* is a primitive type, then *SC* is necessarily the same as *TC*.

The value of the right-hand operand is converted to the type of the selected array component, is subjected to value set conversion (§5.1.13) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the array component.

- ♦ If *TC* is a reference type, then *SC* may not be the same as *TC*, but rather a type that extends or implements *TC*.

Let *RC* be the class of the object referred to by the value of the right-hand operand at run time.

A Java compiler may be able to prove at compile time that the array component will be of type *TC* exactly (for example, *TC* might be `final`). But if a Java compiler cannot prove at compile time that the array component will be of type *TC* exactly, then a check must be performed at run time to ensure that the class *RC* is assignment compatible (§5.2) with the actual type *SC* of the array component.

If class *RC* is not assignable to type *SC*, then no assignment occurs and an `ArrayStoreException` is thrown.

Otherwise, the reference value of the right-hand operand is stored into the selected array component.

Otherwise, three steps are required:

- First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.
- Otherwise, the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the value of the right-hand operand is converted to the type of the left-hand variable, is subjected to value set conversion (§5.1.13) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the variable.

15.26.2 Compound Assignment Operators

A compound assignment expression of the form $E1 \text{ op} = E2$ is equivalent to $E1 = (T) ((E1) \text{ op } (E2))$, where T is the type of $E1$, except that $E1$ is evaluated only once.

At run time, the expression is evaluated in one of two ways.

If the left-hand operand expression is not an array access expression, then:

- First, the left-hand operand is evaluated to produce a variable. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs.
- Otherwise, the value of the left-hand operand is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the saved value of the left-hand variable and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator. If this operation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, the result of the binary operation is converted to the type of the left-hand variable, subjected to value set conversion (§5.1.13) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the variable.

If the left-hand operand expression is an array access expression (§15.13), then:

- First, the array reference subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the index subexpression (of the left-hand operand array access expression) and the right-hand operand are not evaluated and no assignment occurs.
- Otherwise, the index subexpression of the left-hand operand array access expression is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and the right-hand operand is not evaluated and no assignment occurs.
- Otherwise, if the value of the array reference subexpression is `null`, then no assignment occurs and a `NullPointerException` is thrown.
- Otherwise, the value of the array reference subexpression indeed refers to an array. If the value of the index subexpression is less than zero, or greater than or equal to the `length` of the array, then no assignment occurs and an `ArrayIndexOutOfBoundsException` is thrown.
- Otherwise, the value of the index subexpression is used to select a component of the array referred to by the value of the array reference subexpression. The value of this component is saved and then the right-hand operand is evaluated. If this evaluation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.
- Otherwise, consider the array component selected in the previous step, whose value was saved. This component is a variable; call its type *S*. Also, let *T* be the type of the left-hand operand of the assignment operator as determined at compile time.
 - ♦ If *T* is a primitive type, then *S* is necessarily the same as *T*.

The saved value of the array component and the value of the right-hand operand are used to perform the binary operation indicated by the compound assignment operator.

If this operation completes abruptly (the only possibility is an integer division by zero - see §15.17.2), then the assignment expression completes abruptly for the same reason and no assignment occurs.

Otherwise, the result of the binary operation is converted to the type of the selected array component, subjected to value set conversion (§5.1.13) to the appropriate standard value set (not an extended-exponent value set), and the result of the conversion is stored into the array component.

- ♦ If *T* is a reference type, then it must be `String`. Because class `String` is a `final` class, *S* must also be `String`.

The saved value of the array component and the value of the right-hand operand are used to perform the binary operation (string concatenation) indicated by the compound assignment operator (which is necessarily `+=`). If this operation completes abruptly, then the assignment expression completes abruptly for the same reason and no assignment occurs.

Otherwise, the `String` result of the binary operation is stored into the array component.

15.27 Expression

An *Expression* is any assignment expression:

Expression:
AssignmentExpression

Unlike C and C++, the Java programming language has no comma operator.

15.28 Constant Expression

ConstantExpression:
Expression

A compile-time *constant expression* is an expression denoting a value of primitive type or a `String` that does not complete abruptly and is composed using only the following:

- Literals of primitive type and literals of type `String` (§3.10.5)
- Casts to primitive types and casts to type `String`
- The unary operators `+`, `-`, `~`, and `!` (but not `++` or `--`)
- The multiplicative operators `*`, `/`, and `%`
- The additive operators `+` and `-`
- The shift operators `<<`, `>>`, and `>>>`
- The relational operators `<`, `<=`, `>`, and `>=` (but not `instanceof`)

- The equality operators `==` and `!=`
- The bitwise and logical operators `&`, `^`, and `|`
- The conditional-and operator `&&` and the conditional-or operator `||`
- The ternary conditional operator `? :`
- Parenthesized expressions whose contained expression is a constant expression.
- Simple names that refer to constant variables (§4.12.4).
- Qualified names of the form *TypeName* . *Identifier* that refer to constant variables (§4.12.4).

Compile-time constant expressions of type `String` are always "interned" so as to share unique instances, using the method `String.intern`.

A compile-time constant expression is always treated as FP-strict (§15.4), even if it occurs in a context where a non-constant expression would not be considered to be FP-strict.

Definite Assignment

EACH local variable (§14.4) and every blank `final` (§4.12.4) field (§8.3.1.2) must have a *definitely assigned* value when any access of its value occurs. An access to its value consists of the simple name of the variable (or, for a field, the simple name of the field qualified by `this`) occurring anywhere in an expression except as the left-hand operand of the simple assignment operator `=`.

For every access of a local variable or blank `final` field `f`, `f` must be definitely assigned before the access, or a compile-time error occurs.

Similarly, every blank `final` variable must be assigned at most once; it must be *definitely unassigned* when an assignment to it occurs. Such an assignment is defined to occur if and only if either the simple name of the variable (or, for a field, its simple name qualified by `this`) occurs on the left hand side of an assignment operator.

For every assignment to a blank `final` variable, the variable must be definitely unassigned before the assignment, or a compile-time error occurs.

The remainder of this chapter is devoted to a precise explanation of the words "definitely assigned before" and "definitely unassigned before".

The idea behind definite assignment is that an assignment to the local variable or blank `final` field must occur on every possible execution path to the access. Similarly, the idea behind definite unassignment is that no other assignment to the blank `final` variable is permitted to occur on any possible execution path to an assignment.

The analysis takes into account the structure of statements and expressions; it also provides a special treatment of the expression operators `!`, `&&`, `||`, and `?:`, and of boolean-valued constant expressions.

Except for the special treatment of the conditional boolean operators `&&`, `||`, and `? :` and of boolean-valued constant expressions, the values of expressions are not taken into account in the flow analysis.

In order to precisely specify all the cases of definite assignment, the rules in this section define several technical terms:

- whether a variable is *definitely assigned before* a statement or expression;
- whether a variable is *definitely unassigned before* a statement or expression;
- whether a variable is *definitely assigned after* a statement or expression; and
- whether a variable is *definitely unassigned after* a statement or expression.

For boolean-valued expressions, the last two are refined into four cases:

- whether a variable is *definitely assigned after* the expression *when true*;
- whether a variable is *definitely unassigned after* the expression *when true*;
- whether a variable is *definitely assigned after* the expression *when false*; and
- whether a variable is *definitely unassigned after* the expression *when false*.

Here *when true* and *when false* refer to the value of the expression.

The phrase "*V* is definitely assigned after *X*" (where *V* is a local variable and *X* is a statement or expression) means "*V* is definitely assigned after *X* if *X* completes normally". If *X* completes abruptly, the assignment need not have occurred, and the rules stated here take this into account.

The statement "*V* is definitely unassigned after *X*" (where *V* is a variable and *X* is a statement or expression) means "*V* is definitely unassigned after *X* if *X* completes normally".

In all, there are four possibilities for a variable *V* after a statement or expression has been executed:

- *V* is definitely assigned and is not definitely unassigned.
(The flow analysis rules prove that an assignment to *V* has occurred.)
- *V* is definitely unassigned and is not definitely assigned.
(The flow analysis rules prove that an assignment to *V* has not occurred.)
- *V* is not definitely assigned and is not definitely unassigned.
(The rules cannot prove whether or not an assignment to *V* has occurred.)
- *V* is definitely assigned and is definitely unassigned.

(It is impossible for the statement or expression to complete normally.)

16.1 Definite Assignment and Expressions

16.1.1 Boolean Constant Expressions

- V is [un]assigned after any constant expression whose value is `true` when `false`.
- V is [un]assigned after any constant expression whose value is `false` when `true`.
- V is [un]assigned after any constant expression whose value is `true` when `true` iff V is [un]assigned before the constant expression.
- V is [un]assigned after any constant expression whose value is `false` when `false` iff V is [un]assigned before the constant expression.
- V is [un]assigned after a boolean-valued constant expression e iff V is [un]assigned after e when `true` and V is [un]assigned after e when `false`.

16.1.2 The Boolean Operator `&&`

- V is [un]assigned after $a \&\& b$ when `true` iff V is [un]assigned after b when `true`.
- V is [un]assigned after $a \&\& b$ when `false` iff V is [un]assigned after a when `false` and V is [un]assigned after b when `false`.
- V is [un]assigned before a iff V is [un]assigned before $a \&\& b$.
- V is [un]assigned before b iff V is [un]assigned after a when `true`.
- V is [un]assigned after $a \&\& b$ iff V is [un]assigned after $a \&\& b$ when `true` and V is [un]assigned after $a \&\& b$ when `false`.

16.1.3 The Boolean Operator `||`

- V is [un]assigned after $a || b$ when `true` iff V is [un]assigned after a when `true` and V is [un]assigned after b when `true`.
- V is [un]assigned after $a || b$ when `false` iff V is [un]assigned after b when `false`.
- V is [un]assigned before a iff V is [un]assigned before $a || b$.
- V is [un]assigned before b iff V is [un]assigned after a when `false`.

- V is [un]assigned after $a \parallel b$ iff V is [un]assigned after $a \parallel b$ when true and V is [un]assigned after $a \parallel b$ when false.

16.1.4 The Boolean Operator !

- V is [un]assigned after $!a$ when true iff V is [un]assigned after a when false.
- V is [un]assigned after $!a$ when false iff V is [un]assigned after a when true.
- V is [un]assigned before a iff V is [un]assigned before $!a$.
- V is [un]assigned after $!a$ iff V is [un]assigned after $!a$ when true and V is [un]assigned after $!a$ when false.

16.1.5 The Boolean Operator ? :

Suppose that b and c are boolean-valued expressions.

- V is [un]assigned after $a ? b : c$ when true iff V is [un]assigned after b when true and V is [un]assigned after c when true.
- V is [un]assigned after $a ? b : c$ when false iff V is [un]assigned after b when false and V is [un]assigned after c when false.
- V is [un]assigned before a iff V is [un]assigned before $a ? b : c$.
- V is [un]assigned before b iff V is [un]assigned after a when true.
- V is [un]assigned before c iff V is [un]assigned after a when false.
- V is [un]assigned after $a ? b : c$ iff V is [un]assigned after $a ? b : c$ when true and V is [un]assigned after $a ? b : c$ when false.

16.1.6 The Conditional Operator ? :

Suppose that b and c are expressions that are not boolean-valued.

- V is [un]assigned after $a ? b : c$ iff V is [un]assigned after b and V is [un]assigned after c .
- V is [un]assigned before a iff V is [un]assigned before $a ? b : c$.
- V is [un]assigned before b iff V is [un]assigned after a when true.
- V is [un]assigned before c iff V is [un]assigned after a when false.

16.1.7 Other Expressions of Type `boolean`

Suppose that e is an expression of type `boolean` and is not a boolean constant expression, logical complement expression `! a`, conditional-and expression `a && b`, conditional-or expression `a || b`, or conditional expression `a ? b : c`.

- V is [un]assigned after e when true iff V is [un]assigned after e .
- V is [un]assigned after e when false iff V is [un]assigned after e .

16.1.8 Assignment Expressions

Consider an assignment expression `a = b`, `a += b`, `a -= b`, `a *= b`, `a /= b`, `a %= b`, `a <=< b`, `a >>= b`, `a >>= b`, `a &= b`, `a |= b`, or `a ^= b`.

- V is definitely assigned after the assignment expression iff either:
 - ♦ a is V , or
 - ♦ V is definitely assigned after b .
- V is definitely unassigned after the assignment expression iff a is not V and V is definitely unassigned after b .
- V is [un]assigned before a iff V is [un]assigned before the assignment expression.
- V is [un]assigned before b iff V is [un]assigned after a .

16.1.9 Operators `++` and `--`

- V is definitely assigned after `++a`, `--a`, `a++`, or `a--` iff either a is V or V is definitely assigned after the operand expression.
- V is definitely unassigned after `++a`, `--a`, `a++`, or `a--` iff a is not V and V is definitely unassigned after the operand expression.
- V is [un]assigned before a iff V is [un]assigned before `++a`, `--a`, `a++`, or `a--`.

16.1.10 Other Expressions

If an expression is not a boolean constant expression, and is not a preincrement expression `++a`, predecrement expression `--a`, postincrement expression `a++`, postdecrement expression `a--`, logical complement expression `! a`, conditional-and expression `a && b`, conditional-or expression `a || b`, conditional expression `a ? b : c`, or assignment expression, then the following rules apply:

- If the expression has no subexpressions, V is [un]assigned after the expression iff V is [un]assigned before the expression.

This case applies to literals, names, `this` (both qualified and unqualified), unqualified class instance creation expressions with no arguments, initialized array creation expressions whose initializers contain no expressions, unqualified superclass field access expressions, named method invocations with no arguments, and unqualified superclass method invocations with no arguments.

- If the expression has subexpressions, V is [un]assigned after the expression iff V is [un]assigned after its rightmost immediate subexpression.

For any immediate subexpression y of an expression x , V is [un]assigned before y iff one of the following situations is true:

- y is the leftmost immediate subexpression of x and V is [un]assigned before x .
- y is the right-hand operand of a binary operator and V is [un]assigned after the left-hand operand.
- x is an array access, y is the subexpression within the brackets, and V is [un]assigned after the subexpression before the brackets.
- x is a primary method invocation expression, y is the first argument expression in the method invocation expression, and V is [un]assigned after the primary expression that computes the target object.
- x is a method invocation expression or a class instance creation expression; y is an argument expression, but not the first; and V is [un]assigned after the argument expression to the left of y .
- x is a qualified class instance creation expression, y is the first argument expression in the class instance creation expression, and V is [un]assigned after the primary expression that computes the qualifying object.
- x is an array instance creation expression; y is a dimension expression, but not the first; and V is [un]assigned after the dimension expression to the left of y .
- x is an array instance creation expression initialized via an array initializer; y is the array initializer in x ; and V is [un]assigned after the dimension expression to the left of y .

16.2 Definite Assignment and Statements

16.2.1 Empty Statements

- V is [un]assigned after an empty statement iff it is [un]assigned before the empty statement.

16.2.2 Blocks

- A blank `final` member field V is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of any method in the scope of V and before the declaration of any class declared within the scope of V .
- A local variable V is definitely unassigned (and moreover is not definitely assigned) before the block that is the body of the constructor, method, instance initializer or static initializer that declares V .
- Let C be a class declared within the scope of V . Then V is definitely assigned before the block that is the body of any constructor, method, instance initializer, or static initializer declared in C iff V is definitely assigned before the declaration of C .
- V is [un]assigned after an empty block iff V is [un]assigned before the empty block.
- V is [un]assigned after a non-empty block iff V is [un]assigned after the last statement in the block.
- V is [un]assigned before the first statement of the block iff V is [un]assigned before the block.
- V is [un]assigned before any other statement S of the block iff V is [un]assigned after the statement immediately preceding S in the block.

We say that V is definitely unassigned everywhere in a block B iff:

- V is definitely unassigned before B .
- V is definitely assigned after e in every assignment expression $V = e$, $V += e$, $V -= e$, $V *= e$, $V /= e$, $V \% = e$, $V <= e$, $V >= e$, $V >> = e$, $V \& = e$, $V |= e$, or $V \wedge = e$ that occurs in B .
- V is definitely assigned before every expression $++V$, $--V$, $V++$, or $V--$. that occurs in B .

16.2.3 Local Class Declaration Statements

- V is [un]assigned after a local class declaration statement iff V is [un]assigned before the local class declaration statement.

16.2.4 Local Variable Declaration Statements

- V is [un]assigned after a local variable declaration statement that contains no variable initializers iff V is [un]assigned before the local variable declaration statement.
- V is definitely assigned after a local variable declaration statement that contains at least one variable initializer iff either V is definitely assigned after the last variable initializer in the local variable declaration statement or the last variable initializer in the declaration is in the declarator that declares V .
- V is definitely unassigned after a local variable declaration statement that contains at least one variable initializer iff V is definitely unassigned after the last variable initializer in the local variable declaration statement and the last variable initializer in the declaration is not in the declarator that declares V .
- V is [un]assigned before the first variable initializer in a local variable declaration statement iff V is [un]assigned before the local variable declaration statement.
- V is definitely assigned before any variable initializer e other than the first one in the local variable declaration statement iff either V is definitely assigned after the variable initializer to the left of e or the initializer expression to the left of e is in the declarator that declares V .
- V is definitely unassigned before any variable initializer e other than the first one in the local variable declaration statement iff V is definitely unassigned after the variable initializer to the left of e and the initializer expression to the left of e is not in the declarator that declares V .

16.2.5 Labeled Statements

- V is [un]assigned after a labeled statement $L : S$ (where L is a label) iff V is [un]assigned after S and V is [un]assigned before every break statement that may exit the labeled statement $L : S$.
- V is [un]assigned before S iff V is [un]assigned before $L : S$.

16.2.6 Expression Statements

- V is [un]assigned after an expression statement e ; iff it is [un]assigned after e .
- V is [un]assigned before e iff it is [un]assigned before e ;

16.2.7 if Statements

The following rules apply to a statement `if (e) S`:

- V is [un]assigned after `if (e) S` iff V is [un]assigned after S and V is [un]assigned after e when false.
- V is [un]assigned before e iff V is [un]assigned before `if (e) S`.
- V is [un]assigned before S iff V is [un]assigned after e when true.

The following rules apply to a statement `if (e) S else T`:

- V is [un]assigned after `if (e) S else T` iff V is [un]assigned after S and V is [un]assigned after T .
- V is [un]assigned before e iff V is [un]assigned before `if (e) S else T`.
- V is [un]assigned before S iff V is [un]assigned after e when true.
- V is [un]assigned before T iff V is [un]assigned after e when false.

16.2.8 assert Statements

The following rules apply both to a statement `assert e1` and to a statement `assert e1 : e2`:

- V is [un]assigned before $e1$ iff V is [un]assigned before the `assert` statement.
- V is definitely assigned after the `assert` statement iff V is definitely assigned before the `assert` statement.
- V is definitely unassigned after the `assert` statement iff V is definitely unassigned before the `assert` statement and V is definitely unassigned after $e1$ when true.

The following rule applies to a statement `assert e1 : e2`:

- V is [un]assigned before $e2$ iff V is [un]assigned after $e1$ when false.

16.2.9 switch Statements

- V is [un]assigned after a `switch` statement iff all of the following are true:

- ♦ Either there is a `default` label in the switch block and the type of the switch expression is not an enum type; or the type of the switch expression is an enum type and the case labels include all the enum constants of the enum type; or V is [un]assigned after the switch expression.
- ♦ Either there are no switch labels in the switch block that do not begin a block-statement-group (that is, there are no switch labels immediately before the `}` that ends the switch block) or V is [un]assigned after the switch expression.
- ♦ Either the switch block contains no block-statement-groups or V is [un]assigned after the last block-statement of the last block-statement-group.
- ♦ V is [un]assigned before every `break` statement that may exit the switch statement.
- V is [un]assigned before the switch expression iff V is [un]assigned before the switch statement.

If a switch block contains at least one block-statement-group, then the following rules also apply:

- V is [un]assigned before the first block-statement of the first block-statement-group in the switch block iff V is [un]assigned after the switch expression.
- V is [un]assigned before the first block-statement of any block-statement-group other than the first iff V is [un]assigned after the switch expression and V is [un]assigned after the preceding block-statement.

16.2.10 `while` Statements

- V is [un]assigned after `while (e) S` iff V is [un]assigned after e when false and V is [un]assigned before every `break` statement for which the `while` statement is the break target.
- V is definitely assigned before e iff V is definitely assigned before the `while` statement.
- V is definitely unassigned before e iff all of the following conditions hold:
 - ♦ V is definitely unassigned before the `while` statement.
 - ♦ Assuming V is definitely unassigned before e , V is definitely unassigned after S .
 - ♦ Assuming V is definitely unassigned before e , V is definitely unassigned before every `continue` statement for which the `while` statement is the continue target.
- V is [un]assigned before S iff V is [un]assigned after e when true.

16.2.11 do Statements

- V is [un]assigned after $\text{do } S \text{ while } (e)$; iff V is [un]assigned after e when false and V is [un]assigned before every break statement for which the do statement is the break target.
- V is definitely assigned before S iff V is definitely assigned before the do statement.
- V is definitely unassigned before S iff all of the following conditions hold:
 - ♦ V is definitely unassigned before the do statement.
 - ♦ Assuming V is definitely unassigned before S , V is definitely unassigned after e when true.
- V is [un]assigned before e iff V is [un]assigned after S and V is [un]assigned before every continue statement for which the do statement is the continue target.

16.2.12 for Statements

The rules herein cover the basic for statement (§14.14.1). Since the enhanced for (§14.14.2) statement is defined by translation to a basic for statement, no special rules need to be provided for it.

- V is [un]assigned after a for statement iff both of the following are true:
 - ♦ Either a condition expression is not present or V is [un]assigned after the condition expression when false.
 - ♦ V is [un]assigned before every break statement for which the for statement is the break target.
- V is [un]assigned before the initialization part of the for statement iff V is [un]assigned before the for statement.
- V is definitely assigned before the condition part of the for statement iff V is definitely assigned after the initialization part of the for statement.
- V is definitely unassigned before the condition part of the for statement iff all of the following conditions hold:
 - ♦ V is definitely unassigned after the initialization part of the for statement.
 - ♦ Assuming V is definitely unassigned before the condition part of the for statement, V is definitely unassigned after the contained statement.

- ♦ Assuming V is definitely unassigned before the contained statement, V is definitely unassigned before every *continue* statement for which the *for* statement is the *continue* target.
- V is [un]assigned before the contained statement iff either of the following is true:
 - ♦ A condition expression is present and V is [un]assigned after the condition expression when true.
 - ♦ No condition expression is present and V is [un]assigned after the initialization part of the *for* statement.
- V is [un]assigned before the incrementation part of the *for* statement iff V is [un]assigned after the contained statement and V is [un]assigned before every *continue* statement for which the *for* statement is the *continue* target.

16.2.12.1 *Initialization Part*

- If the initialization part of the *for* statement is a local variable declaration statement, the rules of §16.2.4 apply.
- Otherwise, if the initialization part is empty, then V is [un]assigned after the initialization part iff V is [un]assigned before the initialization part.
- Otherwise, three rules apply:
 - ♦ V is [un]assigned after the initialization part iff V is [un]assigned after the last expression statement in the initialization part.
 - ♦ V is [un]assigned before the first expression statement in the initialization part iff V is [un]assigned before the initialization part.
 - ♦ V is [un]assigned before an expression statement S other than the first in the initialization part iff V is [un]assigned after the expression statement immediately preceding S .

16.2.12.2 *Incrementation Part*

- If the incrementation part of the *for* statement is empty, then V is [un]assigned after the incrementation part iff V is [un]assigned before the incrementation part.
- Otherwise, three rules apply:
 - ♦ V is [un]assigned after the incrementation part iff V is [un]assigned after the last expression statement in the incrementation part.

- ♦ V is [un]assigned before the first expression statement in the incrementation part iff V is [un]assigned before the incrementation part.
- ♦ V is [un]assigned before an expression statement S other than the first in the incrementation part iff V is [un]assigned after the expression statement immediately preceding S .

16.2.13 break, continue, return, and throw Statements

- By convention, we say that V is [un]assigned after any break, continue, return, or throw statement.

The notion that a variable is "[un]assigned after" a statement or expression really means "is [un]assigned after the statement or expression completes normally". Because a break, continue, return, or throw statement never completes normally, it vacuously satisfies this notion.

- In a return statement with an expression e or a throw statement with an expression e , V is [un]assigned before e iff V is [un]assigned before the return or throw statement.

16.2.14 synchronized Statements

- V is [un]assigned after synchronized (e) S iff V is [un]assigned after S .
- V is [un]assigned before e iff V is [un]assigned before the statement synchronized (e) S .
- V is [un]assigned before S iff V is [un]assigned after e .

16.2.15 try Statements

These rules apply to every try statement, whether or not it has a finally block:

- V is [un]assigned before the try block iff V is [un]assigned before the try statement.
- V is definitely assigned before a catch block iff V is definitely assigned before the try block.
- V is definitely unassigned before a catch block iff all of the following conditions hold:
 - ♦ V is definitely unassigned after the try block.

- ♦ V is definitely unassigned before every `return` statement that belongs to the `try` block.
- ♦ V is definitely unassigned after e in every statement of the form `throw e` that belongs to the `try` block.
- ♦ V is definitely unassigned after every `assert` statement that occurs in the `try` block.
- ♦ V is definitely unassigned before every `break` statement that belongs to the `try` block and whose `break target` contains (or is) the `try` statement.
- ♦ V is definitely unassigned before every `continue` statement that belongs to the `try` block and whose `continue target` contains the `try` statement.

If a `try` statement does not have a `finally` block, then this rule also applies:

- V is [un]assigned after the `try` statement iff V is [un]assigned after the `try` block and V is [un]assigned after every `catch` block in the `try` statement.

If a `try` statement does have a `finally` block, then these rules also apply:

- V is definitely assigned after the `try` statement iff at least one of the following is true:
 - ♦ V is definitely assigned after the `try` block and V is definitely assigned after every `catch` block in the `try` statement.
 - ♦ V is definitely assigned after the `finally` block.
 - ♦ V is definitely unassigned after a `try` statement iff V is definitely unassigned after the `finally` block.
- V is definitely assigned before the `finally` block iff V is definitely assigned before the `try` statement.
- V is definitely unassigned before the `finally` block iff all of the following conditions hold:
 - ♦ V is definitely unassigned after the `try` block.
 - ♦ V is definitely unassigned before every `return` statement that belongs to the `try` block.
 - ♦ V is definitely unassigned after e in every statement of the form `throw e` that belongs to the `try` block.
 - ♦ V is definitely unassigned after every `assert` statement that occurs in the `try` block.

- ♦ V is definitely unassigned before every `break` statement that belongs to the `try` block and whose `break` target contains (or is) the `try` statement.
- ♦ V is definitely unassigned before every `continue` statement that belongs to the `try` block and whose `continue` target contains the `try` statement.
- ♦ V is definitely unassigned after every `catch` block of the `try` statement.

16.3 Definite Assignment and Parameters

- A formal parameter V of a method or constructor is definitely assigned (and moreover is not definitely unassigned) before the body of the method or constructor.
- An exception parameter V of a `catch` clause is definitely assigned (and moreover is not definitely unassigned) before the body of the `catch` clause.

16.4 Definite Assignment and Array Initializers

- V is [un]assigned after an empty array initializer iff V is [un]assigned before the empty array initializer.
- V is [un]assigned after a non-empty array initializer iff V is [un]assigned after the last variable initializer in the array initializer.
- V is [un]assigned before the first variable initializer of the array initializer iff V is [un]assigned before the array initializer.
- V is [un]assigned before any other variable initializer e of the array initializer iff V is [un]assigned after the variable initializer to the left of e in the array initializer.

16.5 Definite Assignment and Enum Constants

The rules determining when a variable is definitely assigned or definitely unassigned before an enum constant are given in §16.8.

- V is definitely assigned before the declaration of a class body of an enum constant with no arguments that is declared within the scope of V iff V is definitely assigned before the enum constant.

- V is definitely assigned before the declaration of a class body of an enum constant with arguments that is declared within the scope of V iff V is definitely assigned after the last argument expression of the enum constant

The definite assignment/unassignment status of any construct within the class body of an enum constant is governed by the usual rules for classes.

Let y be an argument of an enum constant, but not the first. Then:

- V is [un]assigned before y iff V is [un]assigned after the argument to the left of y .

Otherwise:

- V is [un]assigned before the first argument to an enum constant iff it is [un]assigned before the enum constant.

16.6 Definite Assignment and Anonymous Classes

- V is definitely assigned before an anonymous class declaration (§15.9.5) that is declared within the scope of V iff V is definitely assigned after the class instance creation expression that declares the anonymous class.

16.7 Definite Assignment and Member Types

Let C be a class, and let V be a blank final member field of C . Then:

- V is definitely assigned (and moreover, not definitely unassigned) before the declaration of any member type of C .

Let C be a class declared within the scope of V . Then:

- V is definitely assigned before a member type (§8.5, §9.5) declaration of C iff V is definitely assigned before the declaration of C .

16.8 Definite Assignment and Static Initializers

Let C be a class declared within the scope of V . Then:

- V is definitely assigned before an enum constant or static variable initializer of C iff V is definitely assigned before the declaration of C .

Note that there are no rules that would allow us to conclude that V is definitely unassigned before a static variable initializer or enum constant. We can informally conclude that V is not definitely unassigned before any static variable initializer of C , but there is no need for such a rule to be stated explicitly.

Let C be a class, and let V be a blank `static final` member field of C , declared in C . Then:

- V is definitely unassigned (and moreover is not definitely assigned) before the leftmost enum constant, static initializer, or static variable initializer of C .
- V is [un]assigned before an enum constant, static initializer, or static variable initializer of C other than the leftmost iff V is [un]assigned after the preceding enum constant, static initializer, or static variable initializer of C .

Let C be a class, and let V be a blank `static final` member field of C , declared in a superclass of C . Then:

- V is definitely assigned (and moreover is not definitely unassigned) before every enum constant of C .
- V is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of a static initializer of C .
- V is definitely assigned (and moreover is not definitely unassigned) before every static variable initializer of C .

16.9 Definite Assignment, Constructors, and Instance Initializers

Let C be a class declared within the scope of V . Then:

- V is definitely assigned before an instance variable initializer of C iff V is definitely assigned before the declaration of C .

Note that there are no rules that would allow us to conclude that V is definitely unassigned before an instance variable initializer. We can informally conclude that V is not definitely unassigned before any instance variable initializer of C , but there is no need for such a rule to be stated explicitly.

Let C be a class, and let V be a blank `final non-static` member field of C , declared in C . Then:

- V is definitely unassigned (and moreover is not definitely assigned) before the leftmost instance initializer or instance variable initializer of C .
- V is [un]assigned before an instance initializer or instance variable initializer of C other than the leftmost iff V is [un]assigned after the preceding instance initializer or instance variable initializer of C .

The following rules hold within the constructors of class C :

- V is definitely assigned (and moreover is not definitely unassigned) after an alternate constructor invocation (§8.8.7.1).
- V is definitely unassigned (and moreover is not definitely assigned) before an explicit or implicit superclass constructor invocation (§8.8.7.1).
- If C has no instance initializers or instance variable initializers, then V is not definitely assigned (and moreover is definitely unassigned) after an explicit or implicit superclass constructor invocation.
- If C has at least one instance initializer or instance variable initializer then V is [un]assigned after an explicit or implicit superclass constructor invocation iff V is [un]assigned after the rightmost instance initializer or instance variable initializer of C .

Let C be a class, and let V be a blank final member field of C , declared in a superclass of C . Then:

- V is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of a constructor or instance initializer of C .
- V is definitely assigned (and moreover is not definitely unassigned) before every instance variable initializer of C .

Threads and Locks

WHILE most of the discussion in the preceding chapters is concerned only with the behavior of code as executed a single statement or expression at a time, that is, by a single *thread*, each Java virtual machine can support many threads of execution at once. These threads independently execute code that operates on values and objects residing in a shared main memory. Threads may be supported by having many hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors.

Threads are represented by the `Thread` class. The only way for a user to create a thread is to create an object of this class; each thread is associated with such an object. A thread will start when the `start()` method is invoked on the corresponding `Thread` object.

The behavior of threads, particularly when not correctly synchronized, can be confusing and counterintuitive. This chapter describes the semantics of multithreaded programs; it includes rules for which values may be seen by a read of shared memory that is updated by multiple threads. As the specification is similar to the *memory models* for different hardware architectures, these semantics are known as the *Java programming language memory model*. When no confusion can arise, we will simply refer to these rules as "the memory model".

These semantics do not prescribe how a multithreaded program should be executed. Rather, they describe the behaviors that multithreaded programs are allowed to exhibit. Any execution strategy that generates only allowed behaviors is an acceptable execution strategy.

17.1 Synchronization

The Java programming language provides multiple mechanisms for communicating between threads. The most basic of these methods is *synchronization*, which is implemented using *monitors*. Each object in Java is associated with a monitor, which a thread can *lock* or *unlock*. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread *t* may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation.

The `synchronized` statement (§14.19) computes a reference to an object; it then attempts to perform a lock action on that object's monitor and does not proceed further until the lock action has successfully completed. After the lock action has been performed, the body of the `synchronized` statement is executed. If execution of the body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

A `synchronized` method (§8.4.3.6) automatically performs a lock action when it is invoked; its body is not executed until the lock action has successfully completed. If the method is an instance method, it locks the monitor associated with the instance for which it was invoked (that is, the object that will be known as `this` during execution of the body of the method). If the method is `static`, it locks the monitor associated with the `Class` object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

The Java programming language neither prevents nor requires detection of deadlock conditions. Programs where threads hold (directly or indirectly) locks on multiple objects should use conventional techniques for deadlock avoidance, creating higher-level locking primitives that do not deadlock, if necessary.

Other mechanisms, such as reads and writes of `volatile` variables and the use of classes in the `java.util.concurrent` package, provide alternative ways of synchronization.

17.2 Wait Sets and Notification

Every object, in addition to having an associated monitor, has an associated *wait set*. A wait set is a set of threads.

When an object is first created, its wait set is empty. Elementary actions that add threads to and remove threads from wait sets are atomic. Wait sets are manipulated solely through the methods `Object.wait`, `Object.notify`, and `Object.notifyAll`.

Wait set manipulations can also be affected by the interruption status of a thread, and by the `Thread` class's methods dealing with interruption. Additionally, the `Thread` class's methods for sleeping and joining other threads have properties derived from those of wait and notification actions.

17.2.1 Wait

Wait actions occur upon invocation of `wait()`, or the timed forms `wait(long millisecs)` and `wait(long millisecs, int nanosecs)`.

A thread *returns normally* from a wait if it returns without throwing an `InterruptedException`.

Let thread t be the thread executing the `wait` method on object m , and let n be the number of lock actions by t on m that have not been matched by unlock actions. One of the following actions occurs:

- If n is zero, then an `IllegalMonitorStateException` is thrown.

This is the case where thread t does not already possess the lock for target m .

- If this is a timed wait and the `nanosecs` argument is not in the range of 0–999999 or the `millisecs` argument is negative, then an `IllegalArgumentException` is thrown.
- If thread t is interrupted, then an `InterruptedException` is thrown and t 's interruption status is set to false.
- Otherwise, the following sequence occurs:

1. Thread t is added to the wait set of object m , and performs n unlock actions on m .

Thread t does not execute any further instructions until it has been removed from m 's wait set. The thread may be removed from the wait set due to any one of the following actions, and will resume sometime afterward:

- ♦ A `notify` action being performed on m in which t is selected for removal from the wait set.
- ♦ A `notifyAll` action being performed on m .
- ♦ An `interrupt` action being performed on t .

- ♦ If this is a timed wait, an internal action removing t from m 's wait set that occurs after at least `millisecs` milliseconds plus `nanosecs` nanoseconds elapse since the beginning of this wait action.
- ♦ An internal action by the implementation. Implementations are permitted, although not encouraged, to perform "spurious wake-ups", that is, to remove threads from wait sets and thus enable resumption without explicit instructions to do so.

Each thread must determine an order over the events that could cause it to be removed from a wait set. That order does not have to be consistent with other orderings, but the thread must behave as though those events occurred in that order.

For example, if a thread t is in the wait set for m , and then both an interrupt of t and a notification of m occur, there must be an order over these events. If the interrupt is deemed to have occurred first, then t will eventually return from `wait` by throwing `InterruptedException`, and some other thread in the wait set for m (if any exist at the time of the notification) must receive the notification. If the notification is deemed to have occurred first, then t will eventually return normally from `wait` with an interrupt still pending.

2. Thread t performs n lock actions on m .
3. If thread t was removed from m 's wait set in step 2 due to an interrupt, then t 's interruption status is set to false and the `wait` method throws `InterruptedException`.

17.2.2 Notification

Notification actions occur upon invocation of methods `notify` and `notifyAll`. Let thread t be the thread executing either of these methods on object m , and let n be the number of lock actions by t on m that have not been matched by unlock actions. One of the following actions occurs:

- If n is zero, then an `IllegalMonitorStateException` is thrown.

This is the case where thread t does not already possess the lock for target m .

- If n is greater than zero and this is a `notify` action, then if m 's wait set is not empty, a thread u that is a member of m 's current wait set is selected and removed from the wait set.

There is no guarantee about which thread in the wait set is selected. This removal from the wait set enables u 's resumption in a wait action. Notice, however, that

u's lock actions upon resumption cannot succeed until some time after *t* fully unlocks the monitor for *m*.

- If *n* is greater than zero and this is a `notifyAll` action, then all threads are removed from *m*'s wait set, and thus resume.

Notice, however, that only one of them at a time will lock the monitor required during the resumption of wait.

17.2.3 Interruptions

Interruption actions occur upon invocation of `Thread.interrupt`, as well as methods defined to invoke it in turn, such as `ThreadGroup.interrupt`.

Let *t* be the thread invoking `u.interrupt`, for some thread *u*, where *t* and *u* may be the same. This action causes *u*'s interruption status to be set to true.

Additionally, if there exists some object *m* whose wait set contains *u*, then *u* is removed from *m*'s wait set. This enables *u* to resume in a wait action, in which case this wait will, after re-locking *m*'s monitor, throw `InterruptedException`.

Invocations of `Thread.isInterrupted` can determine a thread's interruption status. The static method `Thread.interrupted` may be invoked by a thread to observe and clear its own interruption status.

17.2.4 Interactions of Waits, Notification, and Interruption

The above specifications allow us to determine several properties having to do with the interaction of waits, notification, and interruption.

If a thread is both notified and interrupted while waiting, it may either:

- return normally from `wait`, while still having a pending interrupt (in other words, a call to `Thread.interrupted` would return true)
- return from `wait` by throwing an `InterruptedException`

The thread may not reset its interrupt status and return normally from the call to `wait`.

Similarly, notifications cannot be lost due to interrupts. Assume that a set *s* of threads is in the wait set of an object *m*, and another thread performs a `notify` on *m*. Then either:

- at least one thread in *s* must return normally from `wait`, or
- all of the threads in *s* must exit `wait` by throwing `InterruptedException`

Note that if a thread is both interrupted and woken via `notify`, and that thread returns from `wait` by throwing an `InterruptedException`, then some other thread in the wait set must be notified.

17.3 Sleep and Yield

`Thread.sleep` causes the currently executing thread to sleep (temporarily cease execution) for the specified duration, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors, and resumption of execution will depend on scheduling and the availability of processors on which to execute the thread.

It is important to note that neither `Thread.sleep` nor `Thread.yield` have any synchronization semantics. In particular, the compiler does not have to flush writes cached in registers out to shared memory before a call to `Thread.sleep` or `Thread.yield`, nor does the compiler have to reload values cached in registers after a call to `Thread.sleep` or `Thread.yield`.

17.4 Memory Model

A *memory model* describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program. The Java programming language memory model works by examining each read in an execution trace and checking that the write observed by that read is valid according to certain rules.

The memory model describes possible behaviors of a program. An implementation is free to produce any code it likes, as long as all resulting executions of a program produce a result that can be predicted by the memory model.

The memory model determines what values can be read at every point in the program. The actions of each thread in isolation must behave as governed by the semantics of that thread, with the exception that the values seen by each read are determined by the memory model. When we refer to this, we say that the program obeys *intra-thread semantics*. Intra-thread semantics are the semantics for single-threaded programs, and allow the complete prediction of the behavior of a thread based on the values seen by read actions within the thread. To determine if the actions of thread *t* in an execution are legal, we simply evaluate the implementation

of thread *t* as it would be performed in a single-threaded context, as defined in the rest of this specification.

Each time the evaluation of thread *t* generates an inter-thread action, it must match the inter-thread action *a* of *t* that comes next in program order. If *a* is a read, then further evaluation of *t* uses the value seen by *a* as determined by the memory model.

This section provides the specification of the Java programming language memory model except for issues dealing with `final` fields, which are described in §17.5.

17.4.1 Shared Variables

Memory that can be shared between threads is called *shared memory* or *heap memory*.

All instance fields, `static` fields, and array elements are stored in heap memory. In this chapter, we use the term *variable* to refer to both fields and array elements.

Local variables (§14.4), formal method parameters (§8.4.1), and exception handler parameters (§14.20) are never shared between threads and are unaffected by the memory model.

Two accesses to (reads of or writes to) the same variable are said to be *conflicting* if at least one of the accesses is a write.

17.4.2 Actions

An *inter-thread action* is an action performed by one thread that can be detected or directly influenced by another thread. There are several kinds of inter-thread action that a program may perform:

- *Read* (normal, or non-volatile). Reading a variable.
- *Write* (normal, or non-volatile). Writing a variable.
- *Synchronization actions*, which are:
 - ♦ *Volatile read*. A volatile read of a variable.
 - ♦ *Volatile write*. A volatile write of a variable.
 - ♦ *Lock*. Locking a monitor
 - ♦ *Unlock*. Unlocking a monitor.
 - ♦ The (synthetic) first and last action of a thread.
 - ♦ Actions that start a thread or detect that a thread has terminated (§17.4.4).

- *External Actions.* An external action is an action that may be observable outside of an execution, and has a result based on an environment external to the execution.
- *Thread divergence actions* (§17.4.9). A thread divergence action is only performed by a thread that is in an infinite loop in which no memory, synchronization, or external actions are performed. If a thread performs a thread divergence action, it will be followed by an infinite number of thread divergence actions.

This specification is only concerned with inter-thread actions. We do not need to concern ourselves with intra-thread actions (e.g., adding two local variables and storing the result in a third local variable). As previously mentioned, all threads need to obey the correct intra-thread semantics for Java programs. We will usually refer to inter-thread actions more succinctly as simply *actions*.

An action a is described by a tuple $\langle t, k, v, u \rangle$, comprising:

- t - the thread performing the action
- k - the kind of action
- v - the variable or monitor involved in the action.

For lock actions, v is the monitor being locked; for unlock actions, v is the monitor being unlocked.

If the action is a (volatile or non-volatile) read, v is the variable being read.

If the action is a (volatile or non-volatile) write, v is the variable being written.

- u - an arbitrary unique identifier for the action

An external action tuple contains an additional component, which contains the results of the external action as perceived by the thread performing the action. This may be information as to the success or failure of the action, and any values read by the action.

Parameters to the external action (e.g., which bytes are written to which socket) are not part of the external action tuple. These parameters are set up by other actions within the thread and can be determined by examining the intra-thread semantics. They are not explicitly discussed in the memory model.

In non-terminating executions, not all external actions are observable. Non-terminating executions and observable actions are discussed in §17.4.9.

17.4.3 Programs and Program Order

Among all the inter-thread actions performed by each thread t , the *program order* of t is a total order that reflects the order in which these actions would be performed according to the intra-thread semantics of t .

A set of actions is *sequentially consistent* if all actions occur in a total order (the execution order) that is consistent with program order, and furthermore, each read r of a variable v sees the value written by the write w to v such that:

- w comes before r in the execution order, and
- there is no other write w' such that w comes before w' and w' comes before r in the execution order.

Sequential consistency is a very strong guarantee that is made about visibility and ordering in an execution of a program. Within a sequentially consistent execution, there is a total order over all individual actions (such as reads and writes) which is consistent with the order of the program, and each individual action is atomic and is immediately visible to every thread.

If a program has no data races, then all executions of the program will appear to be sequentially consistent.

Sequential consistency and/or freedom from data races still allows errors arising from groups of operations that need to be perceived atomically and are not.

17.4.4 Synchronization Order

Every execution has a *synchronization order*. A synchronization order is a total order over all of the synchronization actions of an execution. For each thread t , the synchronization order of the synchronization actions (§17.4.2) in t is consistent with the program order (§17.4.3) of t .

Synchronization actions induce the *synchronized-with* relation on actions, defined as follows:

- An unlock action on monitor m *synchronizes-with* all subsequent lock actions on m (where "subsequent" is defined according to the synchronization order).
- A write to a volatile variable v (§8.3.1.4) *synchronizes-with* all subsequent reads of v by any thread (where "subsequent" is defined according to the synchronization order).
- An action that starts a thread *synchronizes-with* the first action in the thread it starts.

- The write of the default value (zero, false, or null) to each variable *synchronizes-with* the first action in every thread.
- The final action in a thread *T1* *synchronizes-with* any action in another thread *T2* that detects that *T1* has terminated.

T2 may accomplish this by calling *T1.isAlive()* or *T1.join()*.

- If thread *T1* interrupts thread *T2*, the interrupt by *T1* *synchronizes-with* any point where any other thread (including *T2*) determines that *T2* has been interrupted (by having an `InterruptedException` thrown or by invoking `Thread.interrupted` or `Thread.isInterrupted`).

The source of a *synchronizes-with* edge is called a *release*, and the destination is called an *acquire*.

17.4.5 Happens-before Order

Two actions can be ordered by a *happens-before* relationship. If one action *happens-before* another, then the first is visible to and ordered before the second.

If we have two actions *x* and *y*, we write *hb(x, y)* to indicate that *x* *happens-before* *y*.

- If *x* and *y* are actions of the same thread and *x* comes before *y* in program order, then *hb(x, y)*.
- There is a *happens-before* edge from the end of a constructor of an object to the start of a finalizer (§12.6) for that object.
- If an action *x* *synchronizes-with* a following action *y*, then we also have *hb(x, y)*.
- If *hb(x, y)* and *hb(y, z)*, then *hb(x, z)*.

It should be noted that the presence of a *happens-before* relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. If the reordering produces results consistent with a legal execution, it is not illegal.

More specifically, if two actions share a *happens-before* relationship, they do not necessarily have to appear to have happened in that order to any code with which they do not share a *happens-before* relationship. Writes in one thread that are in a data race with reads in another thread may, for example, appear to occur out of order to those reads.

The wait methods of class `Object` (§17.2.1) have lock and unlock actions associated with them; their *happens-before* relationships are defined by these associated actions.

The *happens-before* relation defines when data races take place.

A set of synchronization edges, S , is *sufficient* if it is the minimal set such that the transitive closure of S with the program order determines all of the *happens-before* edges in the execution. This set is unique.

It follows from the above definitions that:

- An unlock on a monitor *happens-before* every subsequent lock on that monitor.
- A write to a `volatile` field (§8.3.1.4) *happens-before* every subsequent read of that field.
- A call to `start()` on a thread *happens-before* any actions in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a `join()` on that thread.
- The default initialization of any object *happens-before* any other actions (other than default-writes) of a program.

When a program contains two conflicting accesses (§17.4.1) that are not ordered by a *happens-before* relationship, it is said to contain a *data race*.

The semantics of operations other than inter-thread actions, such as reads of array lengths (§10.7), executions of checked casts (§5.5, §15.16), and invocations of virtual methods (§15.12), are not directly affected by data races.

A program is *correctly synchronized* if and only if all sequentially consistent executions are free of data races.

If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent (§17.4.3).

We say that a read r of a variable v is allowed to observe a write w to v if, in the *happens-before* partial order of the execution trace:

- r is not ordered before w (i.e., it is not the case that $hb(r, w)$), and
- there is no intervening write w' to v (i.e. no write w' to v such that $hb(w, w')$ and $hb(w', r)$).

Informally, a read r is allowed to see the result of a write w if there is no *happens-before* ordering to prevent that read.

A set of actions A is *happens-before consistent* if for all reads r in A , where $W(r)$ is the write action seen by r , it is not the case that either $hb(r, W(r))$ or that there exists a write w in A such that $w.v = r.v$ and $hb(W(r), w)$ and $hb(w, r)$.

In a *happens-before consistent* set of actions, each read sees a write that it is allowed to see by the *happens-before* ordering.

17.4.6 Executions

An execution E is described by a tuple $\langle P, A, po, so, W, V, sw, hb \rangle$, comprising:

- P - a program
- A - a set of actions
- po - program order, which for each thread t , is a total order over all actions performed by t in A
- so - synchronization order, which is a total order over all synchronization actions in A
- W - a write-seen function, which for each read r in A , gives $W(r)$, the write action seen by r in E .
- V - a value-written function, which for each write w in A , gives $V(w)$, the value written by w in E .
- sw - synchronizes-with, a partial order over synchronization actions
- hb - happens-before, a partial order over actions

Note that the synchronizes-with and happens-before elements are uniquely determined by the other components of an execution and the rules for well-formed executions (§17.4.7).

An execution is *happens-before consistent* if its set of actions is *happens-before consistent* (§17.4.5).

17.4.7 Well-Formed Executions

We only consider well-formed executions. An execution $E = \langle P, A, po, so, W, V, sw, hb \rangle$ is well formed if the following conditions are true:

1. Each read sees a write to the same variable in the execution.

All reads and writes of volatile variables are volatile actions. For all reads r in A , we have $W(r)$ in A and $W(r).v = r.v$. The variable $r.v$ is volatile if and only if r is a volatile read, and the variable $w.v$ is volatile if and only if w is a volatile write.

2. The happens-before order is a partial order.

The happens-before order is given by the transitive closure of synchronizes-with edges and program order. It must be a valid partial order: reflexive, transitive and antisymmetric.

3. The execution obeys intra-thread consistency.

For each thread t , the actions performed by t in A are the same as would be generated by that thread in program-order in isolation, with each write w writing the value $V(w)$, given that each read r sees the value $V(W(r))$. Values seen by each read are determined by the memory model. The program order given must reflect the program order in which the actions would be performed according to the intra-thread semantics of P .

4. The execution is *happens-before consistent* (§17.4.6).
5. The execution obeys synchronization-order consistency.

For all volatile reads r in A , it is not the case that either $so(r, W(r))$ or that there exists a write w in A such that $w.v = r.v$ and $so(W(r), w)$ and $so(w, r)$.

17.4.8 Executions and Causality Requirements

A well-formed execution $E = \langle P, A, po, so, W, V, sw, hb \rangle$ is validated by *committing* actions from A . If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java programming language memory model.

Starting with the empty set as C_0 , we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1} . To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E containing C_i that meets certain conditions.

Formally, an execution E *satisfies the causality requirements of the Java programming language memory model* if and only if there exist:

- Sets of actions $C_0, C_1, \dots$ such that:
 - ♦ C_0 is the empty set
 - ♦ C_i is a proper subset of C_{i+1}
 - ♦ $A = \cup (C_0, C_1, \dots)$

If A is finite, then the sequence $C_0, C_1, \dots$ will be finite, ending in a set $C_n = A$.

If A is infinite, then the sequence $C_0, C_1, \dots$ may be infinite, and it must be the case that the union of all elements of this infinite sequence is equal to A .

- Well-formed executions $E_1, \dots$, where $E_i = \langle P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i \rangle$.

Given these sets of actions $C_0, \dots$ and executions $E_1, \dots$, every action in C_i must be one of the actions in E_i . All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E . Formally:

1. C_i is a subset of A_i
2. $hb_i|_{C_i} = hb|_{C_i}$
3. $so_i|_{C_i} = so|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E . Only the reads in C_{i-1} need to see the same writes in E_i as in E . Formally:

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. Each read r in $C_i - C_{i-1}$ must see writes in C_{i-1} in both E_i and E , but may see a different write in E_i from the one it sees in E . Formally:

6. For any read r in $A_i - C_{i-1}$, we have $hb_i(W_i(r), r)$
7. For any read r in $(C_i - C_{i-1})$, we have $W_i(r)$ in C_{i-1} and $W(r)$ in C_{i-1}

Given a set of sufficient synchronizes-with edges for E_i , if there is a release-acquire pair that happens-before (§17.4.5) an action you are committing, then that pair must be present in all E_j , where $j \geq i$. Formally:

8. Let ssw_i be the sw_i edges that are also in the transitive reduction of hb_i but not in po . We call ssw_i the *sufficient synchronizes-with edges* for E_i . If $ssw_i(x, y)$ and $hb_i(y, z)$ and z in C_i , then $sw_j(x, y)$ for all $j \geq i$.

If an action y is committed, all external actions that happen-before y are also committed.

9. If y is in C_i , x is an external action and $hb_i(x, y)$, then x in C_i .

17.4.9 Observable Behavior and Nonterminating Executions

For programs that always terminate in some bounded finite period of time, their behavior can be understood (informally) simply in terms of their allowable

executions. For programs that can fail to terminate in a bounded amount of time, more subtle issues arise.

The observable behavior of a program is defined by the finite sets of external actions that the program may perform. A program that, for example, simply prints "Hello" forever is described by a set of behaviors that for any non-negative integer i , includes the behavior of printing "Hello" i times.

Termination is not explicitly modeled as a behavior, but a program can easily be extended to generate an additional external action *executionTermination* that occurs when all threads have terminated.

We also define a special *hang* action. If behavior is described by a set of external actions including a *hang* action, it indicates a behavior where after the external actions are observed, the program can run for an unbounded amount of time without performing any additional external actions or terminating. Programs can hang if all threads are blocked or if the program can perform an unbounded number of actions without performing any external actions.

A thread can be blocked in a variety of circumstances, such as when it is attempting to acquire a lock or perform an external action (such as a read) that depends on external data.

An execution may result in a thread being blocked indefinitely and the execution's not terminating. In such cases, the actions generated by the blocked thread must consist of all actions generated by that thread up to and including the action that caused the thread to be blocked, and no actions that would be generated by the thread after that action.

To reason about observable behaviors, we need to talk about sets of observable actions.

If O is a set of observable actions for an execution E , then set O must be a subset of E 's actions, A , and must contain only a finite number of actions, even if A contains an infinite number of actions. Furthermore, if an action y is in O , and either $hb(x, y)$ or $so(x, y)$, then x is in O .

Note that a set of observable actions are not restricted to external actions. Rather, only external actions that are in a set of observable actions are deemed to be observable external actions.

A behavior B is an allowable behavior of a program P if and only if B is a finite set of external actions and either:

- There exists an execution E of P , and a set O of observable actions for E , and B is the set of external actions in O (If any threads in E end in a blocked state and O contains all actions in E , then B may also contain a *hang* action); or
- There exists a set O of actions such that B consists of a *hang* action plus all the external actions in O and for all $k \geq |O|$, there exists an execution E of P with actions A , and there exists a set of actions O' such that:
 - ♦ Both O and O' are subsets of A that fulfill the requirements for sets of observable actions.
 - ♦ $O \subseteq O' \subseteq A$
 - ♦ $|O'| \geq k$
 - ♦ $O' - O$ contains no external actions

17.5 final Field Semantics

Fields declared `final` are initialized once, but never changed under normal circumstances. The detailed semantics of `final` fields are somewhat different from those of normal fields. In particular, compilers have a great deal of freedom to move reads of `final` fields across synchronization barriers and calls to arbitrary or unknown methods. Correspondingly, compilers are allowed to keep the value of a `final` field cached in a register and not reload it from memory in situations where a non-`final` field would have to be reloaded.

`final` fields also allow programmers to implement thread-safe immutable objects without synchronization. A thread-safe immutable object is seen as immutable by all threads, even if a data race is used to pass references to the immutable object between threads. This can provide safety guarantees against misuse of an immutable class by incorrect or malicious code. `final` fields must be used correctly to provide a guarantee of immutability.

An object is considered to be *completely initialized* when its constructor finishes. A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's `final` fields.

The usage model for `final` fields is a simple one: Set the `final` fields for an object in that object's constructor; and do not write a reference to the object being constructed in a place where another thread can see it before the object's constructor is finished. If this is followed, then when the object is seen by another thread, that

thread will always see the correctly constructed version of that object's final fields. It will also see versions of any object or array referenced by those final fields that are at least as up-to-date as the final fields are.

17.5.1 Semantics of final Fields

The semantics for final fields are as follows. Let o be an object, and c be a constructor for o in which a final field f is written. A *freeze* action on final field f of o takes place when c exits, either normally or abruptly.

Note that if one constructor invokes another constructor, and the invoked constructor sets a final field, the freeze for the final field takes place at the end of the invoked constructor.

For each execution, the behavior of reads is influenced by two additional partial orders, the dereference chain *dereferences()* and the memory chain *mc()*, which are considered to be part of the execution (and thus, fixed for any particular execution). These partial orders must satisfy the following constraints (which need not have a unique solution):

- **Dereference Chain:** If an action a is a read or write of a field or element of an object o by a thread t that did not initialize o , then there must exist some read r by thread t that sees the address of o such that r *dereferences*(r, a).
- **Memory Chain:** There are several constraints on the memory chain ordering:
 - ♦ If r is a read that sees a write w , then it must be the case that $mc(w, r)$.
 - ♦ If r and a are actions such that *dereferences*(r, a), then it must be the case that $mc(r, a)$.
 - ♦ If w is a write of the address of an object o by a thread t that did not initialize o , then there must exist some read r by thread t that sees the address of o such that $mc(r, w)$.

Given a write w , a freeze f , an action a (that is not a read of a final field), a read r_1 of the final field frozen by f , and a read r_2 such that $hb(w, f)$, $hb(f, a)$, $mc(a, r_1)$, and *dereferences*(r_1, r_2), then when determining which values can be seen by r_2 , we consider $hb(w, r_2)$. (This *happens-before* ordering does not transitively close with other *happens-before* orderings.)

Note that the *dereferences* order is reflexive, and r_1 can be the same as r_2 .

For reads of final fields, the only writes that are deemed to come before the read of the final field are the ones derived through the final field semantics.

17.5.2 Reading final Fields During Construction

A read of a final field of an object within the thread that constructs that object is ordered with respect to the initialization of that field within the constructor by the usual *happens-before* rules. If the read occurs after the field is set in the constructor, it sees the value the final field is assigned, otherwise it sees the default value.

17.5.3 Subsequent Modification of final Fields

In some cases, such as deserialization, the system will need to change the final fields of an object after construction. final fields can be changed via reflection and other implementation-dependent means. The only pattern in which this has reasonable semantics is one in which an object is constructed and then the final fields of the object are updated. The object should not be made visible to other threads, nor should the final fields be read, until all updates to the final fields of the object are complete. Freezes of a final field occur both at the end of the constructor in which the final field is set, and immediately after each modification of a final field via reflection or other special mechanism.

Even then, there are a number of complications. If a final field is initialized to a compile-time constant in the field declaration, changes to the final field may not be observed, since uses of that final field are replaced at compile time with the compile-time constant.

Another problem is that the specification allows aggressive optimization of final fields. Within a thread, it is permissible to reorder reads of a final field with those modifications of a final field that do not take place in the constructor.

An implementation may provide a way to execute a block of code in a *final-field-safe context*. If an object is constructed within a final-field-safe context, the reads of a final field of that object will not be reordered with modifications of that final field that occur within that final-field-safe context.

A final-field-safe context has additional protections. If a thread has seen an incorrectly published reference to an object that allows the thread to see the default value of a final field, and then, within a final-field-safe context, reads a properly published reference to the object, it will be guaranteed to see the correct value of the final field. In the formalism, code executed within a final-field-safe context is treated as a separate thread (for the purposes of final field semantics only).

In an implementation, a compiler should not move an access to a final field into or out of a final-field-safe context (although it can be moved around the execution of such a context, so long as the object is not constructed within that context).

17.5.4 Write-protected Fields

Normally, a field that is `final` and `static` may not be modified. However, `System.in`, `System.out`, and `System.err` are `static final` fields that, for legacy reasons, must be allowed to be changed by the methods `System.setIn`, `System.setOut`, and `System.setErr`. We refer to these fields as being *write-protected* to distinguish them from ordinary `final` fields.

The compiler needs to treat these fields differently from other `final` fields. For example, a read of an ordinary `final` field is "immune" to synchronization: the barrier involved in a lock or `volatile` read does not have to affect what value is read from a `final` field. Since the value of write-protected fields may be seen to change, synchronization events should have an effect on them. Therefore, the semantics dictate that these fields be treated as normal fields that cannot be changed by user code, unless that user code is in the `System` class.

17.6 Word Tearing

One consideration for implementations of the Java virtual machine is that every field and array element is considered distinct; updates to one field or element must not interact with reads or updates of any other field or element. In particular, two threads that update adjacent elements of a byte array separately must not interfere or interact and do not need synchronization to ensure sequential consistency.

Some processors do not provide the ability to write to a single byte. It would be illegal to implement byte array updates on such a processor by simply reading an entire word, updating the appropriate byte, and then writing the entire word back to memory. This problem is sometimes known as *word tearing*, and on processors that cannot easily update a single byte in isolation some other approach will be required.

17.7 Non-atomic Treatment of `double` and `long`

For the purposes of the Java programming language memory model, a single write to a non-volatile `long` or `double` value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Writes and reads of volatile long and double values are always atomic. Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.

Syntax

THIS chapter presents a grammar for the Java programming language.

The grammar presented piecemeal in the preceding chapters is much better for exposition, but it is not well suited as a basis for a parser. The grammar presented in this chapter is the basis for the reference implementation. Note that it is not an LL(1) grammar, though in many cases it minimizes the necessary look ahead.

The grammar below uses the following BNF-style conventions:

- $[x]$ denotes zero or one occurrences of x .
- $\{x\}$ denotes zero or more occurrences of x .
- $x \mid y$ means one of either x or y .

Identifier:

IDENTIFIER

QualifiedIdentifier:

Identifier { . Identifier }

QualifiedIdentifierList:

QualifiedIdentifier { , QualifiedIdentifier }

CompilationUnit:

```
[ [Annotations] package QualifiedIdentifier ; ]
    {ImportDeclaration} {TypeDeclaration}
```

ImportDeclaration:

```
import [static] Identifier { . Identifier } [ . * ] ;
```

TypeDeclaration:

```
ClassOrInterfaceDeclaration
;

```

ClassOrInterfaceDeclaration:

```
{Modifier} (ClassDeclaration | InterfaceDeclaration)
```

ClassDeclaration:

```
NormalClassDeclaration
EnumDeclaration

```

InterfaceDeclaration:

```
NormalInterfaceDeclaration
AnnotationTypeDeclaration

```

NormalClassDeclaration:

```
class Identifier [TypeParameters] [extends Type] [implements TypeList] ClassBody
```

EnumDeclaration:

```
enum Identifier [implements TypeList] EnumBody
```

NormalInterfaceDeclaration:

```
interface Identifier [TypeParameters] [extends TypeList] InterfaceBody
```

AnnotationTypeDeclaration:

```
@ interface Identifier AnnotationTypeBody
```

Type:

ReferenceType { [] }

BasicType { [] }

ReferenceType:

Identifier [*TypeArguments*] { . *Identifier* [*TypeArguments*] }

TypeArguments:

< *TypeArgument* { , *TypeArgument* } >

TypeArgument:

ReferenceType

? [(extends | super) *ReferenceType*]

BasicType:

byte

short

char

int

long

float

double

boolean

TypeParameters:

< *TypeParameter* { , *TypeParameter* } >

TypeParameter:

Identifier [extends *Bound*]

Bound:

ReferenceType { & *ReferenceType* }

NonWildcardTypeArguments:

< *TypeList* >

TypeList:

ReferenceType { , *ReferenceType* }

Modifier:

Annotation
 public
 protected
 private
 static
 abstract
 final
 native
 synchronized
 transient
 volatile
 strictfp

Annotations:

Annotation {Annotation}

Annotation:

@ *QualifiedIdentifier* [([*AnnotationElement*])]

AnnotationElement:

ElementValuePairs
ElementValue

ElementValuePairs:

ElementValuePair { , *ElementValuePair* }

ElementValuePair:

Identifier = *ElementValue*

ElementValue:

Annotation
Expression1
ElementValueArrayInitializer

ElementValueArrayInitializer:

{ [*ElementValues*] [,] }

ElementValues:

ElementValue { , *ElementValue* }

ClassBody:
 { { *ClassBodyDeclaration* } }

ClassBodyDeclaration:
 ;
 {*Modifier*} *MemberDecl*
 [*static*] *Block*

MemberDecl:
MethodOrFieldDecl
 void *Identifier* *VoidMethodDeclaratorRest*
Identifier *ConstructorDeclaratorRest*
GenericMethodOrConstructorDecl
ClassDeclaration
InterfaceDeclaration

MethodOrFieldDecl:
Type *Identifier* *MethodOrFieldRest*

MethodOrFieldRest:
VariableDeclaratorsRest ;
MethodDeclaratorRest

MethodDeclaratorRest:
FormalParameters { [] } [*throws QualifiedIdentifierList*] (*Block* | ;)

VoidMethodDeclaratorRest:
FormalParameters [*throws QualifiedIdentifierList*] (*Block* | ;)

ConstructorDeclaratorRest:
FormalParameters [*throws QualifiedIdentifierList*] *Block*

GenericMethodOrConstructorDecl:
TypeParameters *GenericMethodOrConstructorRest*

GenericMethodOrConstructorRest:
 (*Type* | void) *Identifier* *MethodDeclaratorRest*
Identifier *ConstructorDeclaratorRest*

InterfaceBody:

{ { InterfaceBodyDeclaration } }

InterfaceBodyDeclaration:

*;
{Modifier} InterfaceMemberDecl*

InterfaceMemberDecl:

*InterfaceMethodOrFieldDecl
void Identifier VoidInterfaceMethodDeclaratorRest
InterfaceGenericMethodDecl
ClassDeclaration
InterfaceDeclaration*

InterfaceMethodOrFieldDecl:

Type Identifier InterfaceMethodOrFieldRest

InterfaceMethodOrFieldRest:

*ConstantDeclaratorsRest ;
InterfaceMethodDeclaratorRest*

ConstantDeclaratorsRest:

ConstantDeclaratorRest { , ConstantDeclarator }

ConstantDeclaratorRest:

{ [] } = VariableInitializer

ConstantDeclarator:

Identifier ConstantDeclaratorRest

InterfaceMethodDeclaratorRest:

FormalParameters { [] } [throws QualifiedIdentifierList] ;

VoidInterfaceMethodDeclaratorRest:

FormalParameters [throws QualifiedIdentifierList] ;

InterfaceGenericMethodDecl:

TypeParameters (Type | void) Identifier InterfaceMethodDeclaratorRest

FormalParameters:

([*FormalParameterDecls*])

FormalParameterDecls:

{*VariableModifier*} *Type FormalParameterDeclsRest*

VariableModifier:

final

Annotation

FormalParameterDeclsRest:

VariableDeclaratorId [, *FormalParameterDecls*]

... *VariableDeclaratorId*

VariableDeclaratorId:

Identifier { [] }

VariableDeclarators:

VariableDeclarator { , *VariableDeclarator* }

VariableDeclarator:

Identifier VariableDeclaratorRest

VariableDeclaratorRest:

{ [] } [= *VariableInitializer*]

VariableInitializer:

ArrayInitializer

Expression

ArrayInitializer:

{ [*VariableInitializer* { , *VariableInitializer* } [,]] }

VariableDeclaratorsRest:

VariableDeclaratorRest { , *VariableDeclarator* }

Block:

{ *BlockStatements* }

BlockStatements:

{ *BlockStatement* }

BlockStatement:

LocalVariableDeclarationStatement

ClassOrInterfaceDeclaration

[*Identifier* :] *Statement*

LocalVariableDeclarationStatement:

{ *VariableModifier* } *Type* *VariableDeclarators* ;

Statement:

Block

assert *Expression* [: *Expression*] ;

if *ParExpression* *Statement* [else *Statement*]

while *ParExpression* *Statement*

do *Statement* while *ParExpression* ;

synchronized *ParExpression* *Block*

return [*Expression*] ;

throw *Expression* ;

break [*Identifier*] ;

continue [*Identifier*] ;

try *Block* (*Catches* | [*Catches*] finally *Block*)

switch *ParExpression* { *SwitchBlockStatementGroups* }

for (*ForControl*) *Statement*

;

StatementExpression ;

Identifier : *Statement*

StatementExpression:

Expression

Catches:

CatchClause { *CatchClause* }

CatchClause:

catch ({*VariableModifier*} *Type* *Identifier*) *Block*

SwitchBlockStatementGroups:
 { *SwitchBlockStatementGroup* }

SwitchBlockStatementGroup:
 SwitchLabels *BlockStatements*

SwitchLabels:
 SwitchLabel { *SwitchLabel* }

SwitchLabel:
 case *Expression* :
 case *EnumConstantName* :
 default :

EnumConstantName:
 Identifier

ForControl:
 ForVarControl
 ForInit ; [*Expression*] ; [*ForUpdate*]

ForVarControl:
 {*VariableModifier*} *Type* *VariableDeclaratorId* *ForVarControlRest*

ForVarControlRest:
 ForVariableDeclaratorsRest ; [*Expression*] ; [*ForUpdate*]
 : *Expression*

ForVariableDeclaratorsRest:
 [= *VariableInitializer*] { , *VariableDeclarator* }

ForInit:
ForUpdate:
 StatementExpression { , *StatementExpression* }

Expression:

Expression1 [AssignmentOperator Expression1]

AssignmentOperator:

=

+=

-=

**=*

/=

&=

|=

^=

%=

<<=

>>=

>>>=

Expression1:

Expression2 [Expression1Rest]

Expression1Rest:

? Expression : Expression1

Expression2:

Expression3 [Expression2Rest]

Expression2Rest:

{ InfixOp Expression3 }

instanceof Type

SYNTAX

InfixOp:

||
&&
|
^
&
==
!=
<
>
<=
>=
<<
>>
>>>
+
-
*
/
%

Expression3:

PrefixOp Expression3
(Expression | Type) Expression3
Primary { Selector } { PostfixOp }

PrefixOp:

++
--
!
~
+
-

PostfixOp:

++
--

*Primary:**Literal**ParExpression**this [Arguments]**super SuperSuffix**new Creator**NonWildcardTypeArguments (ExplicitGenericInvocationSuffix | this Arguments)**Identifier { . Identifier } [IdentifierSuffix]**BasicType { [] } . class**void . class**Literal:**IntegerLiteral**FloatingPointLiteral**CharacterLiteral**StringLiteral**BooleanLiteral**NullLiteral**ParExpression:**(Expression)**Arguments:**([Expression { , Expression }])**SuperSuffix:**Arguments**. Identifier [Arguments]**ExplicitGenericInvocationSuffix:**super SuperSuffix**Identifier Arguments*

Creator:

*NonWildcardTypeArguments CreatedName ClassCreatorRest
CreatedName (ClassCreatorRest | ArrayCreatorRest)*

CreatedName:

Identifier [TypeArguments] { . Identifier [TypeArguments] }

ClassCreatorRest:

Arguments [ClassBody]

ArrayCreatorRest:

*[
 ([{}] } ArrayInitializer |
 Expression [{ [Expression] } { {} }]
]*

IdentifierSuffix:

*[({ [] } . class | Expression)]
Arguments
 . (class | ExplicitGenericInvocation | this | super Arguments |
 new [NonWildcardTypeArguments] InnerCreator)*

ExplicitGenericInvocation:

NonWildcardTypeArguments ExplicitGenericInvocationSuffix

InnerCreator:

Identifier ClassCreatorRest

Selector:

*. Identifier [Arguments]
 . ExplicitGenericInvocation
 . this
 . super SuperSuffix
 . new [NonWildcardTypeArguments] InnerCreator
 [Expression]*

EnumBody:
 { [*EnumConstants*] [,] [*EnumBodyDeclarations*] }

EnumConstants:
EnumConstant
EnumConstants , *EnumConstant*

EnumConstant:
 [*Annotations*] *Identifier* [*Arguments*] [*ClassBody*]

EnumBodyDeclarations:
 ; {*ClassBodyDeclaration*}

AnnotationTypeBody:
 { [*AnnotationTypeElementDeclarations*] }

AnnotationTypeElementDeclarations:
AnnotationTypeElementDeclaration
AnnotationTypeElementDeclarations *AnnotationTypeElementDeclaration*

AnnotationTypeElementDeclaration:
 {*Modifier*} *AnnotationTypeElementRest*

AnnotationTypeElementRest:
Type *Identifier* *AnnotationMethodOrConstantRest* ;
ClassDeclaration
InterfaceDeclaration
EnumDeclaration
AnnotationTypeDeclaration

AnnotationMethodOrConstantRest:
AnnotationMethodRest
ConstantDeclaratorsRest

AnnotationMethodRest:
 () [[]] [*default ElementValue*]